TURING 图灵程序设计丛书

Packt>

Hands-On Meta Learning with Python

# Python元学习

## 通用人工智能的实现

[印] 苏达桑·拉维尚迪兰◎著　　葛言◎译



- 用Python语言全面解析各种单样本学习算法及其实现
- 学习先进的元学习算法，真正实现通用人工智能

 中国工信出版集团　 人民邮电出版社 POSTS & TELECOM PRESS

作者

**苏达桑·拉维尚迪兰**（Sudharsan Ravichandiran）

目前在众包网站Freelancer担任数据科学家。他是积极的开源项目贡献者和畅销书作家，重点关注深度学习和强化学习的实际应用，尤其是自然语言处理和计算机视觉领域的相关研究。

译者

**葛言**

本科毕业于华中科技大学经济学院国际商务专业（英语双学位），保送上海财经大学交叉科学研究院管理科学与工程直博，目前从事运筹学相关研究与Python开发工作。译有《精通Python设计模式（第2版）》等书。

# 数字版权声明


图灵程序设计丛书

Hands-On Meta Learning with Python

# Python元学习

## 通用人工智能的实现

[印] 苏达桑 · 拉维尚迪兰◎著　　葛言◎译

人民邮电出版社
北　京

图书在版编目（CIP）数据

Python元学习 : 通用人工智能的实现 / (印) 苏达桑·拉维尚迪兰著 ; 葛言译. -- 北京 : 人民邮电出版社, 2020.6
（图灵程序设计丛书）
ISBN 978-7-115-53967-0

Ⅰ. ①P… Ⅱ. ①苏… ②葛… Ⅲ. ①软件工具—程序设计 Ⅳ. ①TP311.561

中国版本图书馆CIP数据核字(2020)第077912号

## 内 容 提 要

元学习是当今人工智能研究的热门领域之一，被视为实现通用人工智能的基础。本书介绍元学习及其原理，讲解各种单样本学习算法，例如孪生网络、原型网络、关系网络和记忆增强网络，并在基于Python的TensorFlow与Keras中实现它们。读者能够从本书中了解先进的元学习算法，如模型无关元学习、Reptile和元学习的上下文适应。此外，本书还探索如何使用元随机梯度下降法来快速学习，以及如何使用元学习来进行无监督学习。

本书适合机器学习爱好者、人工智能研究人员和数据科学家阅读。

◆ 著　[印] 苏达桑·拉维尚迪兰
译　葛　言
责任编辑　温　雪
责任印制　周昇亮

◆ 人民邮电出版社出版发行　北京市丰台区成寿寺路11号
邮编　100164　电子邮件　315@ptpress.com.cn
网址　https://www.ptpress.com.cn
北京　印刷

◆ 开本：800×1000　1/16
印张：9.75
字数：231千字　2020年6月第1版
印数：1－2 500册　2020年6月北京第1次印刷

著作权合同登记号　图字：01-2019-4616号

定价：59.00元

读者服务热线：(010)51095183转600　印装质量热线：(010)81055316
反盗版热线：(010)81055315
广告经营许可证：京东市监广登字 20170147 号

# 版权声明

本书献给我慈爱的母亲卡苏里、敬爱的父亲拉维尚迪兰。

——苏达桑·拉维尚迪兰

# 前　　言

本书解释元学习的基本原理，并帮助你理解“学会学习”（learning to learn）的概念。你将学习各种单样本学习算法，例如孪生网络（siamese network）、原型网络（prototypical network）、关系网络（relation network）和记忆增强网络（memory-augmented network），并在 TensorFlow 与 Keras 中实现它们；了解先进的元学习算法，如模型无关元学习（model-agnostic meta learning，MAML）、Reptile 和元学习的上下文适应（context adaptation via meta learning，CAML）；探索如何使用元随机梯度下降法（meta stochastic gradient descent，Meta-SGD）来快速学习，以及如何使用元学习来进行无监督学习。

## 本书读者

本书主要写给那些想了解元学习并将其作为先进方法来训练机器学习模型的机器学习爱好者、人工智能研究人员和数据科学家。读者应当具备机器学习实战经验和坚实的 Python 编程知识。

## 本书内容

**第 1 章，元学习简介**，帮助你理解什么是元学习，并介绍它的不同类型。我们将研究元学习如何通过学习少量数据点来进行少样本学习，然后熟悉梯度下降（gradient descent）以及少样本学习的模型优化。

**第 2 章，使用孪生网络进行人脸识别与音频识别**，首先解释什么是孪生网络，以及如何在单样本学习中使用孪生网络，然后研究孪生网络的架构及其一些应用，以及如何使用孪生网络来建立人脸识别模型与音频识别模型。

**第 3 章，原型网络及其变体**，解释什么是原型网络以及如何在少样本学习中使用它。我们将建立一个对 omniglot 字符集进行分类的原型网络，并学习原型网络的不同变体，如高斯原型网络和半原型网络。

**第 4 章，使用 TensorFlow 构建关系网络与匹配网络**，帮助你理解关系网络的架构，及其在单样本、少样本和零样本学习中的用途。我们将介绍如何使用 TensorFlow 构建关系网络并将学习匹配网络及其架构，还将探索完整的上下文嵌入，以及如何使用 TensorFlow 构建匹配网络。

**第 5 章，记忆增强神经网络**，介绍神经图灵机（NTM）的概念，以及 NTM 如何利用外部存储空间存储和检索信息。我们将研究 NTM 中使用的不同寻址机制，并了解记忆增强神经网络（MANN）及其与 NTM 架构的区别。

**第 6 章，MAML 及其变种**，介绍一种流行的元学习算法——MAML。我们将探索什么是 MAML，如何在监督和强化学习环境中使用它，以及如何从头构建它。此外还会学习对抗性元学习和 CAML，其中后者用于元学习中的快速上下文适应。

**第 7 章，Meta-SGD 和 Reptile**，解释如何使用 Meta-SGD 学习梯度下降算法的所有内容，如初始权重、学习率和更新方向。我们将了解如何从头开始构建 Meta-SGD；学习 Reptile 算法，它可以被视为 MAML 的改进版；掌握如何使用 Reptile 算法进行正弦曲线回归。

**第 8 章，梯度一致作为优化目标**，介绍如何在元学习环境中使用梯度一致作为优化目标。我们将学习什么是梯度一致及其如何增强元学习算法，还将学习如何从头构建梯度一致算法。

**第 9 章，新进展与未来方向**，首先解释什么是任务无关元学习；然后介绍如何在模仿学习中使用元学习，以及如何使用 CACTUs 算法将 MAML 应用到无监督学习中；最后在概念空间中探索一种叫作“学会学习”的深度元学习算法。

## 如何充分利用本书

你需要安装下列软件：

- Python
- Anaconda
- TensorFlow
- Keras

## 下载示例代码

你可以用你的账户从 www.packtpub.com 下载已购买的 Packt 图书的示例代码文件。如果你是从其他地方购买的本书，可以访问 www.packtpub.com/support 并注册，我们将通过电子邮件把文件发送给你。[①]

你可以通过以下步骤下载本书示例代码：

(1) 在 www.packtpub.com 登录或注册；
(2) 选择 SUPPORT 标签；

① 示例代码也可以从本书图灵社区页面（https://www.ituring.com.cn/book/2697）获取。——编者注

(3) 点击 Code Downloads & Errata；
(4) 在 Search 框输入书名并遵循屏幕上的指示。

下载完文件后，确保使用如下软件的最新版本来解压或提取文件夹。

- Windows 上建议使用 WinRAR/7-Zip。
- Mac 上建议使用 Zipeg/iZip/UnRarX。
- Linux 上建议使用 7-Zip/PeaZip。

本书的代码包也可在 GitHub 上获取，在 GitHub 网站搜索 Hands-On-Meta-Learning-with-Python 即可。如果代码更新了，也会在现有的 GitHub 仓库上更新。

GitHub 网站 PackPublishing 页面上也列出了我们所出版的各类图书和视频的代码包。欢迎查看！

## 排版约定

本书采用如下排版约定。

文本中的代码采用等宽字体，例如：“`read_image` 函数将一张图片作为输入，并返回一个 NumPy 数组。”

代码块的格式如下：

```
import re
import numpy as np
from PIL import Image
```

**黑体字**：表示新术语、重要的词，或界面上显示的内容。



此图标表示警告或重要信息。




此图标表示提示或诀窍。

## 保持联系

我们始终欢迎读者的反馈。

**一般反馈**：如果你对本书有任何疑问，请通过 questions@packtpub.com联系我们，并在邮件主题中注明书名。

**勘误**：虽然我们已经竭尽全力来确保本书内容的准确性，但错误在所难免。如果你发现书中有错，请告知我们，我们将不胜感激。请访问 www.packtpub.com/submit-errata，选择图书，点击勘误提交表单链接，并输入详细信息。

**反盗版**：如果你在网上发现以任何形式复制我们作品的非法行为，请立即将地址或网站名称告知我们，我们将不胜感激。请联系 copyright@packtpub.com提供有盗版嫌疑的链接。

**成为作者**：如果你是某个领域的专家，并且有兴趣编写图书，请访问 authors.packtpub.com。

## 评论

请留下你的评论。阅读并使用本书之后，为何不在购买网站上发表评论呢？其他读者可以参考你的评价来做出购买决定，Packt 出版社可以了解你对我们产品的看法，作者也可以看到你对本书的反馈。谢谢！

想了解更多关于 Packt 的信息，请访问 packt.com。

## 电子书

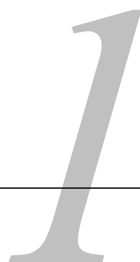
扫描如下二维码，即可购买本书中文版电子版。



# 目　　录

第 1 章

# 元学习简介



元学习是目前人工智能领域最有前景和最热门的研究方向之一，被视为实现**通用人工智能**（Artificial General Intelligence，AGI）的基础。本章，我们将学习什么是元学习，以及为什么元学习是目前人工智能领域最令人兴奋的研究方向；了解什么是少样本、单样本和零样本学习，以及它们在元学习中的应用；学习不同类型的元学习技术；探索“通过梯度下降来学习如何通过梯度下降来学习”（learning to learn by gradient descent by gradient descent）的概念，以及如何使用元学习器学习梯度下降优化；了解如何将优化作为一个模型用于少样本学习，并看到如何在少样本学习中将元学习器用作优化算法。

本章内容包括：

- 元学习；
- 元学习与少样本学习；
- 元学习的类型；
- 通过梯度下降来学习如何通过梯度下降来学习；
- 少样本学习中的优化模型。

## 1.1 元学习

元学习是目前人工智能领域中一个令人振奋的研究方向。随着大量研究论文的发表和研究进展的取得，元学习在人工智能领域取得了重大突破。在开始探讨元学习之前，先来了解一下当前的人工智能模型的工作原理。

近年来，随着生成对抗网络和胶囊网络等优秀算法的出现，深度学习得到了快速的发展。但问题是，深度神经网络需要大规模的训练集来训练模型。当数据点很少时，它会突然失效。假设我们训练了一个深度学习模型来执行任务 A。当我们有一个和 A 紧密相关的新任务 B 时，就不能使用相同的模型，而是需要从零开始为任务 B 训练模型。因此，虽然每个任务可能是相关的，但都需要从零开始训练模型。

深度学习真的是真正的人工智能吗？答案是否定的。我们人类是如何学习的呢？我们将学到的东西归纳为多个概念并从中学习。不过目前的学习算法只能处理一项任务。这就是元学习的用武之地。元学习能够生成一个通用的人工智能模型来学习执行各种任务，而无须从零开始训练它们。我们可以用很少的数据点来训练元学习模型去完成各种相关的任务，因此对于一个新任务，元学习模型可以利用之前从相关任务中获得的知识，无须从零开始训练。许多研究人员和科学家认为，元学习可以让我们更接近 AGI。接下来的几节将阐释元学习模型如何学会学习的过程。

### 元学习与少样本学习

**少样本学习**（few-shot learning）或 ***k*** **样本学习**（*k*-shot learning）指的是利用较少的数据点进行学习，其中 $k$ 表示数据集各个类别中数据点的数量。假设我们正在对狗和猫的图像进行分类。如果只有 1 张狗的图像和 1 张猫的图像，那就叫作**单样本学习**（one-shot learning）。也就是说，对于每个类别，我们只从 1 个数据点学习。如果有 10 张狗的图像和 10 张猫的图像，那就叫作 10 样本学习。因此，$k$ 样本学习中的 $k$ 表示每个类别中数据点的数量。还有一种**零样本学习**（zero-shot learning），即所有类别中都没有数据点。等等，如果没有数据点，那可怎么学习呢？在这种情况下，我们虽然没有数据点，但是拥有关于每个类别的元信息，并将从中学习。因为数据集中有两个类别，即狗和猫，所以可以称之为双（$n=2$）类别 $k$ 样本学习——$n$ 表示数据集中类别的数量。

为了使模型从少量的数据点中学习，我们将用同样的方法训练它们。因此，当有一个数据集 $D$ 时，我们从数据集中的每个类别中挑选几个数据点，称之为**支撑集**（support set）。同样，从每个类别中挑选一些不同的数据点，称之为**查询集**（query set）。于是，我们用一个支撑集训练模型，并用查询集来测试模型。我们以一种**阶段式的方式**（episodic fashion）训练模型，即在每个阶段中，从数据集 $D$ 中抽取少量数据点，准备支撑集和查询集，并使用支撑集进行训练，使用查询集进行测试。因此，在多个阶段后，模型将学会如何从较小的数据集中学习。接下来的章节会对此进行更详细的探讨。

## 1.2　元学习的类型

元学习有多种分类标准，例如寻找最优的权重集与学习优化器。我们将元学习分为以下 3 类：

- 学习度量空间
- 学习初始化
- 学习优化器

### 1.2.1　学习度量空间

在基于度量的元学习场景中，我们将学习合适的度量空间。假设我们想学习两幅图像之间的相似性。在基于度量的场景中，我们使用一个简单的神经网络从两幅图像中提取特征，并通过计

算两幅图像特征之间的距离找到相似性。这种方法被广泛应用于数据点较少的少样本学习中。接下来的章节将介绍基于度量的学习算法，如孪生网络、原型网络和关系网络。

### 1.2.2　学习初始化

在这个方法中，我们尝试学习最优的初始参数值。这是什么意思呢？假设我们正在构建一个神经网络来对图像进行分类。我们首先初始化随机权重，计算损失，并通过梯度下降来最小化损失。因此，我们将通过梯度下降找到最优权重，使损失最小。如果不随机初始化权重，而是用最优值或者接近最优值的值来初始化权重，那么就可以更快地收敛，并快速学习。接下来的章节将介绍如何通过 MAML、Reptile 和 Meta-SGD 等算法来精确地找到这些最优的初始权重。

### 1.2.3　学习优化器

在这个方法中，我们尝试学习优化器。一般如何优化神经网络呢？答案是通过基于大数据集的训练来优化神经网络，并使用梯度下降来最小化损失。但是在少样本的学习场景中，梯度下降失效了，因为我们的数据集较小。因此，在这种情况下，我们将学习优化器本身。我们将有两个网络：试图学习的基网络和优化基网络的元网络。后面的章节会详细探讨。

## 1.3　通过梯度下降来学习如何通过梯度下降来学习

现在，我们来看一个有趣的元学习算法——通过梯度下降来学习如何通过梯度下降来学习。这个名字是不是有点吓人？实际上，它是最简单的元学习算法之一。我们知道，在元学习中，目标是学习学习的过程。一般如何训练神经网络呢？答案是通过梯度下降来计算损失和最小化损失以训练网络。因此，我们使用梯度下降法来优化模型。如果不使用梯度下降，我们能自动学习这个优化过程吗？

但是应该如何学习呢？我们用**递归神经网络**（Recurrent Neural Network，RNN）代替传统的梯度下降优化器。这是如何实现的呢？如何用 RNN 代替梯度下降法？如果你仔细观察梯度下降的行为，就会发现，它基本上是一个从输出层到输入层的更新序列。我们将这些更新存储在一个状态中，这样就可以使用 RNN 并将更新存储在 RNN 单元中。

该算法的主要思想是用 RNN 代替梯度下降法。但问题是 RNN 如何学习？如何优化 RNN？为了优化 RNN，我们使用梯度下降法。简而言之，**我们正在学习通过 RNN 来执行梯度下降，而这个 RNN 是通过梯度下降来优化的。**这就是该算法名称的由来。

我们称 RNN 为优化器，称基网络（base network）为优化对象。假设有一个由参数 $\theta$ 影响的模型 $f$。需要找到这个最优参数 $\theta$，以将损失最小化。一般情况下，通过梯度下降法来寻找最优参数，但现在用 RNN 来寻找最优参数。因此，RNN（优化器）找到了最优参数并将其发送给优

化对象（基网络）。优化对象使用这个参数，计算损失，并将损失发送给 RNN。基于该损失，RNN 通过梯度下降优化自身，并更新模型参数 $\theta$ 。

感到困惑吗？请看图 1-1：优化对象（基网络）是通过优化器（RNN）优化的。优化器将更新后的参数（即权重）发送给优化对象，优化对象使用这些权重计算损失，并将损失发送给优化器。基于损失，优化器通过梯度下降来改进自身。

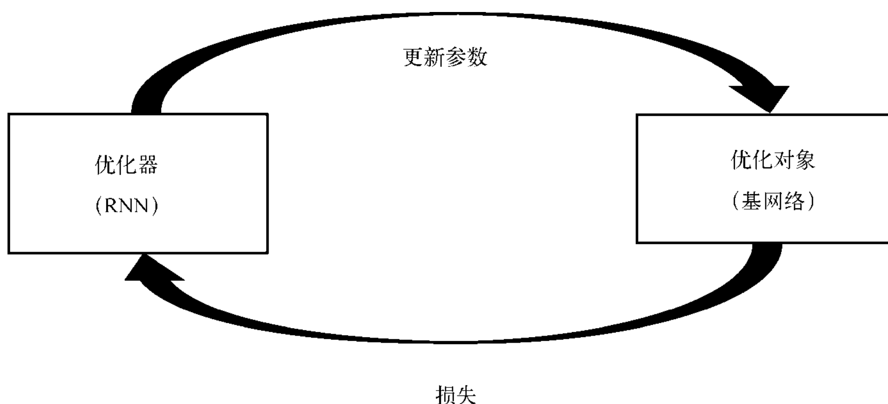



图　1-1

假设基网络（优化对象）以 $\theta$ 作为参数，而 RNN（优化器）以 $\phi$ 作为参数。优化器的损失函数是什么？我们知道优化器（RNN）用于减少优化对象（基网络）的损失。因此，优化器的损失是优化对象的平均损失，它可以表示为

$$L(\phi)=\mathbb{E}_f[f(\theta(f,\phi))]$$

怎样才能把损失降到最低呢？通过梯度下降找到合适的 $\phi$ 来最小化这种损失。RNN 接受什么作为输入？它又输出什么呢？优化器，也就是 RNN，将优化对象的梯度 $\nabla_t$ 以及它的上一个状态 $h_t$ 作为输入，并返回输出——可以最小化优化器损失的更新 $g_t$ 。我们用函数 $m$ 来表示 RNN：

$$(g_t,h_{t+1})=m(\nabla_t,h_t,\phi)$$

以上方程的参数解释如下：

- ❑ $\nabla_t$ 是模型 $f$ （优化对象）的梯度，即 $\nabla_t=\nabla_t f(\theta_t)$ ；
- ❑ $h_t$ 是 RNN 的隐藏状态；
- ❑ $\phi$ 是 RNN 的参数；
- ❑ 输出 $g_t$ 和 $h_{t+1}$ 分别是（提供给优化器的）更新与 RNN 的下一个状态。

于是，可以使用 $\theta_{t+1}=\theta_t+g_t$ 来更新模型参数值。

如图 1-2 所示，优化器 $m$ 在时间 $t$ 处，以隐藏状态 $h_t$ 和相对于 $\theta_t$ 的梯度 $\nabla_t$ 为输入，计算出 $g_t$ 并将其发送到优化对象，再与 $\theta_t$ 相加，成为下一步更新的 $\theta_{t+1}$。

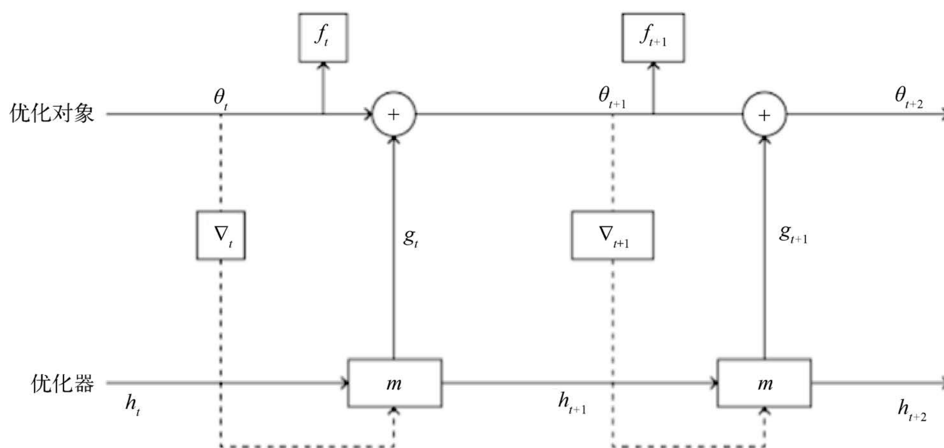



图　1-2

由此，我们通过梯度下降学会了梯度下降优化。

## 1.4　少样本学习的优化模型

我们知道，少样本学习基于较少的数据点，那么如何将梯度下降应用到少样本学习中呢？在少样本学习中，梯度下降会由于数据点非常少而突然失效。梯度下降优化需要更多的数据点来达到收敛和损失最小化。因此，在少样本学习中需要一种更好的优化技术。假设有一个由参数 $\theta$ 影响的模型 $f$。我们用一些随机值来初始化参数 $\theta$，并尝试使用梯度下降法找到最优值。让我们回忆一下梯度下降的更新方程：

$$\theta_t = \theta_{t-1} - \alpha_t \nabla_{\theta_{t-1}} L_t$$

以上方程的参数解释如下：

- $\theta_t$ 是更新参数；
- $\theta_{t-1}$ 是上一步的参数值；
- $\alpha_t$ 是学习率；
- $\nabla_{\theta_{t-1}} L_t$ 是相对于 $\theta_{t-1}$ 的损失函数的梯度。

梯度下降的更新方程是不是看起来很熟悉？是的，你猜对了，它类似于长短期记忆网络（LSTM）的细胞状态更新方程，可以写成：

$$c_t = f_t \odot c_{t-1} + i_t \odot \tilde{c}_t$$

可以将 LSTM 细胞更新方程与梯度下降完全对应起来，设 $f_t = 1$，可得：

$$c_{t-1} = \theta_{t-1}$$
$$i_t = \alpha_t$$
$$\tilde{c}_t = \nabla_{\theta_{t-1}} L_t$$

因此，在少样本学习中，可以使用 LSTM 而非梯度下降作为优化器。LSTM 是元学习器，它将学习用于训练模型的更新规则。因此，我们使用两个网络：一个是基学习器，它学会执行任务；另一个是元学习器，它试图找到最优的参数。这是如何实现的呢？

我们知道，LSTM 使用遗忘门（forget gate）来丢弃存储器中不需要的信息，它可以表示为

$$f_t = \sigma(w_f \cdot [h_{t-1}, x_t] + b_f)$$

这个遗忘门在我们的优化场景中有什么用呢？假设我们处在一个损失很大，梯度接近于零的位置。怎样才能摆脱这种局面呢？在这种情况下，可以收缩模型的参数，并忘记其前一个值的某些部分。我们可以使用遗忘门来实现这一点，它以当前参数值 $\theta_{t-1}$、当前损失 $L_t$、当前梯度 $\nabla_{\theta_{t-1}}$ 以及前一个遗忘门作为输入。它可以表示为

$$f_t = \sigma(w_f \cdot [\theta_{t-1}, L_t, \nabla_{\theta_{t-1}}, f_{t-1}] + b_f)$$

下面来看看输入门（input gate）。我们知道 LSTM 中的输入门是用来决定更新什么值的，它可以表示为

$$i_t = \sigma(w_i \cdot [h_{t-1}, x_t] + b_i)$$

在少样本学习中，可以使用这个输入门来调整学习率，从而在防止发散的同时快速学习：

$$i_t = \sigma(w_i \cdot [\theta_{t-1}, L_t, \nabla_{\theta_{t-1}}, i_{t-1}] + b_i)$$

因此，元学习器在多次更新之后得到了 $i_t$ 与 $f_t$ 的最优值。

可是，这是如何运作的呢？

假设有一个由 $\theta$ 影响的基网络 $M$、由 $\phi$ 影响的 LSTM 元学习器 $R$，以及数据集 $D$。我们将数据集分割为训练集 $D^{\text{train}}$ 和测试集 $D^{\text{test}}$。首先随机初始化元学习器参数 $\phi$。

在 $T$ 次迭代中，随机从 $D^{\text{train}}$ 中抽取数据点，计算损失以及相对于模型参数 $\theta$ 的损失梯度。将这个梯度、损失和元学习器参数 $\phi$ 提供给元学习器。元学习器 $R$ 会返回细胞状态 $c_t$，然后在时间 $t$ 将基网络 $M$ 的参数 $\theta_t$ 更新为 $c_t$。重复 $N$ 次，如图 1-3 所示。

for $t = 1, \cdots, T$ do

$X_t, Y_t \longleftarrow D^{\text{train}}$中的一批随机样本

$\text{Loss}_t \longleftarrow L(M(X_t; \theta_{t-1}), Y_t)$

细胞状态（$c_t$）$\longleftarrow R((\nabla_{\theta_{t-1}} \text{Loss}_t, \text{Loss}_t), \phi)$

$\theta_t \longleftarrow c_t$

end for

图　1-3

因此，经过 $T$ 次迭代，我们会得到一个最优参数 $\theta_T$。不过如何检查 $\theta_T$ 的性能并更新元学习器参数呢？使用测试集和参数 $\theta_T$ 计算测试集的损失。然后，计算相对于元学习器参数 $\phi$ 的损失梯度，并更新 $\phi$，如图 1-4 所示。

$X, Y \longleftarrow D^{\text{test}}$

$\text{Loss}_{\text{test}} \longleftarrow L(M(X; \theta_T), Y)$

用$\nabla_\phi \text{Loss}_{\text{test}}$更新$\phi$

图　1-4

迭代 $n$ 次，并更新元学习器。完整的算法如图 1-5 所示。

$\phi_0 \longleftarrow$ 随机初始化

for $d = 1, \cdots, n$ do

$D^{\text{train}}, D^{\text{test}} \longleftarrow$ 数据集 $D$ 中的随机样本

$\theta_0 \longleftarrow c_0$

for $t = 1, \cdots, T$ do

$X_t, Y_t \longleftarrow D^{\text{train}}$ 中的一批随机样本

$\text{Loss}_t \longleftarrow L(M(X_t; \theta_{t-1}), Y_t)$

细胞状态（$c_t$）$\longleftarrow R((\nabla_{\theta_{t-1}} \text{Loss}_t, \text{Loss}_t), \phi_{d-1})$

$\theta_t \longleftarrow c_t$

end for

$X, Y \longleftarrow D^{\text{test}}$

$\text{Loss}_{\text{test}} \longleftarrow L(M(X; \theta_T), Y_t)$

用$\nabla_{\theta_{t-1}} \text{Loss}_{\text{test}}$更新$\phi_d$

end for

图　1-5

## 1.5 小结

我们首先学习了什么是元学习，以及如何在元学习中使用单样本学习、少样本学习和零样本学习，并了解到支撑集和查询集与训练集和测试集十分相似，只不过每个类别中都有 $k$ 个数据点，还知道了 $n$ 类别 $k$ 样本学习的含义。然后，我们了解了不同类型的元学习技术，探讨了通过梯度下降来学习如何通过梯度下降来学习，并知道了如何使用 RNN 作为优化器优化基网络。之后，我们将优化看作一个用于少样本学习的模型，并将 LSTM 用作元学习器在少样本学习中进行优化。

## 1.6 思考题

(1) 什么是元学习？
(2) 什么是少样本学习？
(3) 什么是支撑集？
(4) 什么是查询集？
(5) 基于度量的学习又称为什么？
(6) 如何在元学习中进行训练？

## 1.7 延伸阅读

- 通过梯度下降来学习如何通过梯度下降来学习：参见 Marcin Andrychowicz、Misha Denil、Sergio Gomez 等人的文章 *Learning to Learn by Gradient Descent by Gradient Descent*。
- 少样本学习场景下的优化模型：参见 Sachin Ravi 和 Hugo Larochelle 的文章 *Optimization as a Model for Few-shot Learning Setting*。

第 2 章

# 使用孪生网络进行人脸识别与音频识别

在上一章，我们学习了什么是元学习以及有哪些不同类型的元学习技术，介绍了通过梯度下降来学习如何通过梯度下降来学习，并将优化作为少样本学习的一个模型。在本章，我们将学习一种十分常用的基于度量的单样本学习算法——孪生网络，了解它们如何从很少的数据点中学习，以及如何使用它们来解决低数据问题（low data problem）；随后详细探讨孪生网络的架构，并了解孪生网络的一些应用；最后学习如何使用孪生网络建立人脸识别模型和音频识别模型。

本章内容包括：

- 什么是孪生网络；
- 孪生网络的架构；
- 孪生网络的应用；
- 使用孪生网络进行人脸识别；
- 使用孪生网络建立音频识别模型。

## 2.1 什么是孪生网络

孪生网络是一种特殊的神经网络，是最简单、最常用的单样本学习算法之一。正如第 1 章所说，单样本学习在每个类别中只学习一个训练实例。因此，孪生网络主要用于各类别数据点较少的应用中。例如，我们想为组织建立一个人脸识别模型，而组织中大约有 500 人。如果我们想用**卷积神经网络**（Convolutional Neural Network，CNN）从零开始建立人脸识别模型，那么需要这 500 个人的很多图像来训练网络并获得良好的精度，但显然我们不会有这 500 个人的太多图像，因此除非有足够的数据点，否则使用 CNN 或任何深度学习算法来建立模型并不可行。这种情况下，可以使用复杂的单样本学习算法来解决问题，比如孪生网络，它可以从较少的数据点中学习。

但孪生网络是如何运作的呢？孪生网络大致上由两个对称的神经网络组成，它们具有相同的权重和架构，并在最后由能量函数 $E$ 连接在一起。我们的孪生网络的目标是了解两个输入值是否相似。假设有两幅图像 $X_1$ 和 $X_2$，我们想知道这两幅图像是否相似。

如图 2-1 所示，我们将图像 $X_1$ 提供给网络 A，将图像 $X_2$ 提供给网络 B。这两个网络的作用都是为输入图像生成嵌入（embedding），即特征向量（feature vector）。因此，我们可以使用任意产生嵌入的网络。因为输入是一个图像，所以可以使用卷积网络来生成嵌入，也就是提取特征。记住，CNN 在这里的作用只是提取特征而不是分类。我们知道这些网络应该有相同的权重和架构，如果网络 A 是一个三层 CNN，那么网络 B 也应该是一个三层 CNN，我们必须为这两个网络使用相同的权重。因此，网络 A 和网络 B 将分别给出输入图像 $X_1$ 和 $X_2$ 的嵌入。然后，我们将这些嵌入提供给能量函数，它会给出这两个输入的相似程度。能量函数基本上可以是任何相似性度量，如欧氏距离和余弦相似度。

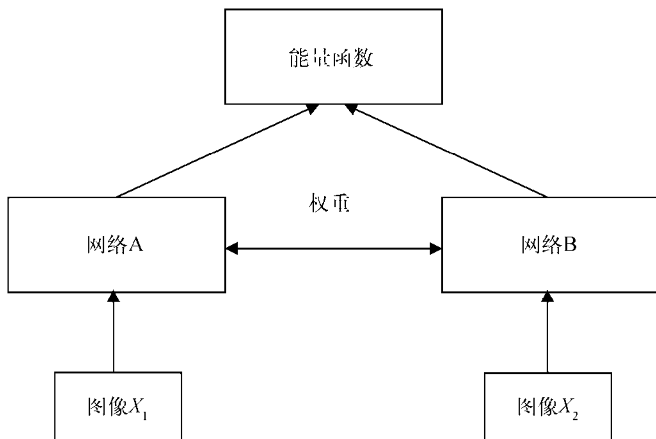



图　2-1

孪生网络不仅用于人脸识别，而且还广泛用于数据点较少的应用以及需要学习两个输入之间相似性的任务中。孪生网络的应用包括签名验证、相似问题检索、目标跟踪等。下一节将详细研究孪生网络。

## 2.1.1　孪生网络的架构

对孪生网络有了基本的了解后，我们将对其进行详细介绍。它的架构如图 2-2 所示。

如图 2-2 所示，孪生网络由两个相同的网络组成，它们具有相同的权重和架构。假设有两个输入，$X_1$ 和 $X_2$。把输入 $X_1$ 输入网络 A，也就是 $f_w(X_1)$，并把输入 $X_2$ 输入网络 B，也就是 $f_w(X_2)$。你会注意到，这两个网络有相同的权重 $\boldsymbol{w}$，它们会为输入（$X_1$ 和 $X_2$）生成嵌入。然后，将这些

嵌入提供给能量函数 $E$，即可得到两个输入之间的相似性。

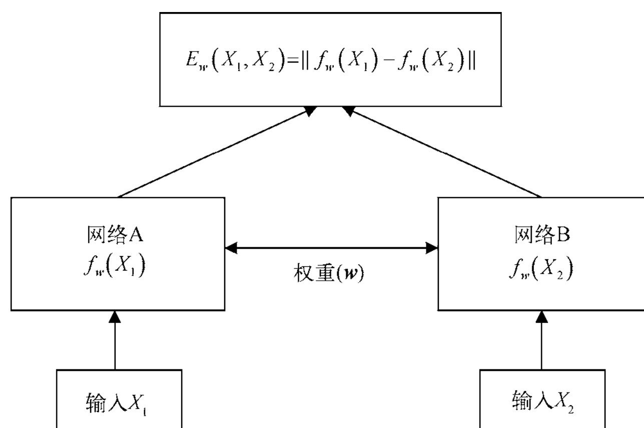



图 2-2

表达式如下：

$$E_w(X_1,X_2)=\|f_w(X_1)-f_w(X_2)\|$$

假设我们用欧氏距离作为能量函数，如果 $X_1$ 和 $X_2$ 相似，那么 $E$ 的值会较小。如果输入值不相似，那么 $E$ 的值会很大。

假设有两个句子，句子 1 和句子 2。将句子 1 输入网络 A，句子 2 输入网络 B。假设网络 A 和网络 B 都是 LSTM 网络，且权重相同。因此，网络 A 和网络 B 将分别为句子 1 和句子 2 生成词嵌入。然后，将这些嵌入输入能量函数，从而得到两个句子之间的相似度。但是如何训练孪生网络呢？数据应该是怎样的？特征和标签是什么？我们的目标函数是什么？

孪生网络的输入($X_1,X_2$)应该成对出现，它们的二元标签（binary label）$Y\in\{0,1\}$ 代表输入对是正样本对（genuine pair，相同）还是负样本对（imposite pair，不同）。我们有如表 2-1 所示的句子对，从标签可以看出句子对是正（1）还是负（0）：

表 2-1

| 句子对 | | 标签 |
|---|---|---|
| She is a beautiful girl | She is a gorgeous girl | 1 |
| Birds fly in the sky | What are you doing | 0 |
| I love Paris | I adore Paris | 1 |
| He just arrived | I am watching a movie | 0 |

那么，孪生网络的损失函数是什么呢？由于孪生网络的目标不是执行分类任务，而是理解两个输入值之间的相似性，因此我们使用了对比损失函数。

它的表达式如下：

$$\text{对比损失} = Y(E)^2 + (1-Y)\max(\text{margin} - E, 0)^2$$

在上式中，$Y$ 代表真实标签（true label），当两个输入值相似时为 1，当两个输入值不相似时为 0。$E$ 是能量函数，它可以是任何距离度量。变量 margin 用于保存约束，也就是说，当两个输入值不相似时，如果它们的距离大于 margin 的值，那么就不会导致损失。

### 2.1.2　孪生网络的应用

正如我们所理解的，孪生网络通过使用相同的架构在两个输入值之间寻找相似性来学习。在涉及计算两个实体之间相似性的任务中，它是最常用的少样本学习算法之一。它功能强大、稳健，可以作为低数据问题的解决方案。

在关于孪生网络的第一篇论文（*Signature Verification Using a "Siamese" Time Delay Neural Network*，作者为 Jane Bromley、Isabelle Guyon、Yann LeCun 等人）中，作者阐述了孪生网络对于签名验证任务的意义。签名验证任务的目标是识别签名的真实性，因此，作者用正样本签名对和负样本签名对训练了孪生网络，利用卷积网络从签名中提取了特征，然后通过测量两个特征向量之间的距离来识别相似性。当一个新的签名出现时，提取这些特征并将它们与存储的签名者特征向量进行比较，如果距离小于某个阈值，则接受签名是真实的，反之则拒绝签名。

孪生网络也广泛应用于自然语言处理（NLP）任务中。在一篇有趣的论文 *Learning Text Similarity with Siamese Recurrent Networks* 中，作者 Paul Neculoiu、Maarten Versteegh 和 Mihai Rotaru 使用孪生网络计算文本相似度。他们使用孪生网络作为双向单元，使用余弦相似度作为能量函数，来计算文本之间的相似度。

孪生网络的应用范围相当广泛，它们可以组装在各种架构中，用于执行各种任务，比如人类动作识别、场景变化检测和机器翻译等。

## 2.2　使用孪生网络进行人脸识别

我们将通过建立人脸识别模型来了解孪生网络。我们的孪生网络的目的是理解两张脸相似与否。我们将使用 AT&T 的人脸数据库。

下载并解压 AT&T 的人脸数据库文件后，可以看到 s1、s2 一直到 s40 文件夹，如图 2-3 所示。

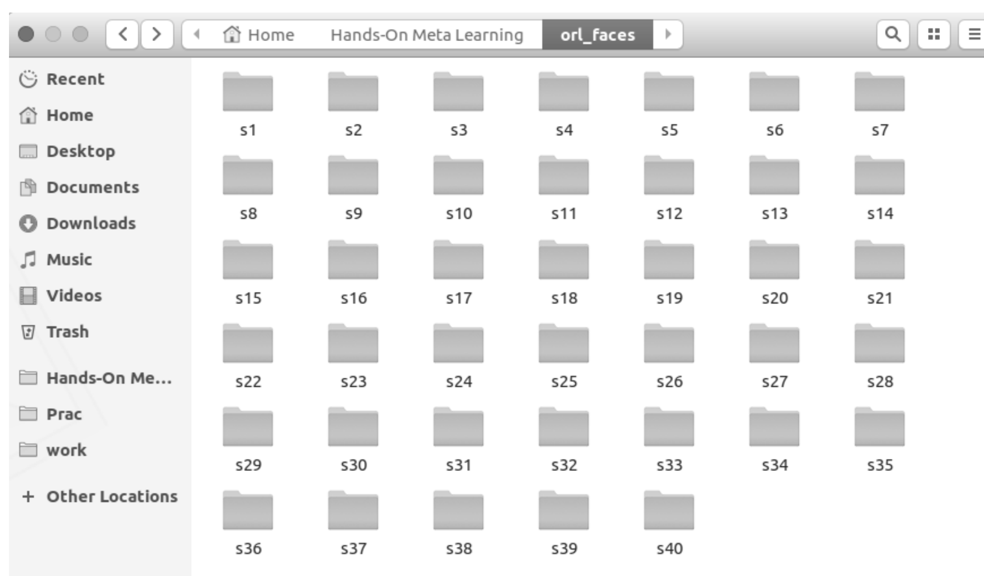



图　2-3

每个文件夹都有同一个人从不同角度拍摄的 10 张照片。例如，打开文件夹 s1，可以看到同一个人的 10 张不同的照片，如图 2-4 所示。

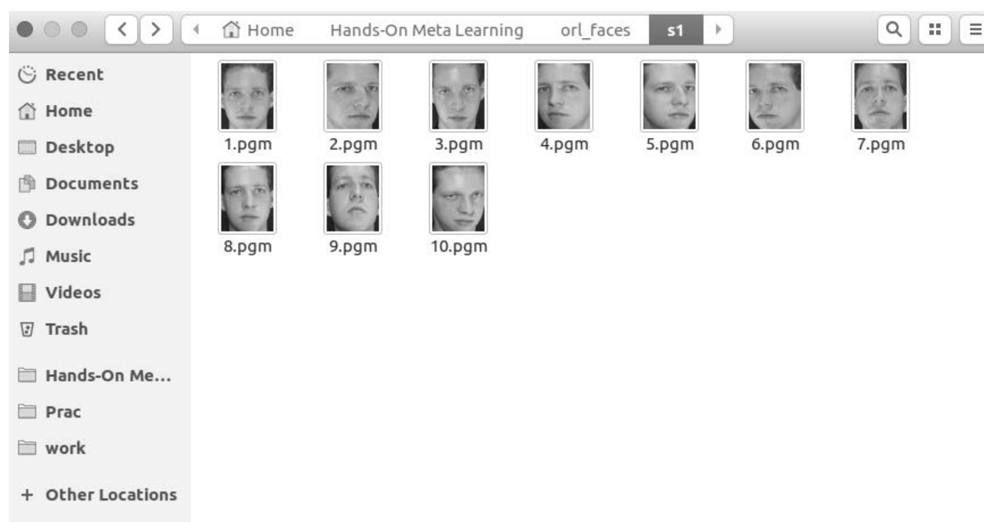



图　2-4

我们打开并检查文件夹 s13，如图 2-5 所示。

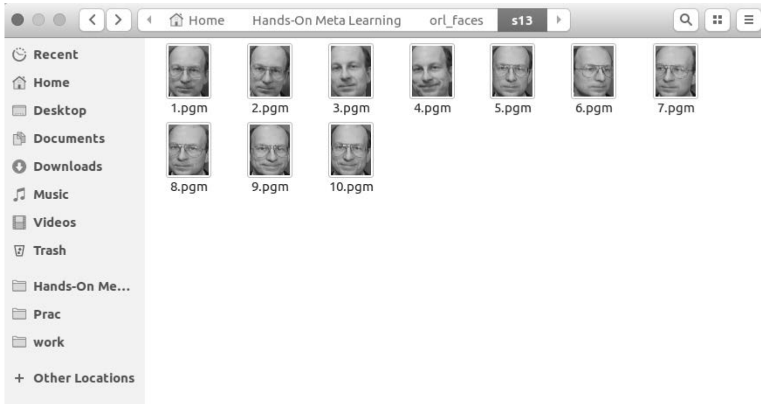



图　2-5

因为我们知道孪生网络要求输入值成对并带有标签，所以必须以这种方式创建数据。我们从同一个文件夹中随机取出两张图像，并将它们标记为正样本对；从两个文件夹中分别取出一张图像，并将它们标记为负样本对。如图 2-6 所示，正样本对的照片上是同一个人，而负样本对的照片上是不同的人。

| 输　入　对 | | 标　　签 |
|---|---|---|
| | | 正 |
| | | 负 |
| | | 正 |
| | | 负 |

图　2-6

有了成对的数据以及标签，就可以训练孪生网络了。我们将图像对中的一个图像输入网络 A，另一个图像输入网络 B。这两个网络的作用只是提取特征向量。我们使用带有**整流线性单元**（rectified linear unit，ReLU）激活函数的两个卷积层提取特征。一旦学习了特征，我们就把两个网络输出的特征向量输入能量函数，用于测量相似度（用欧氏距离作为能量函数）。因此，我们通过输入图像对来训练网络，以学习它们之间的语义相似度。下面，我们一步一步来分析。

2

为了更好地理解，可以在 GitHub 网站搜索 Hands-On-Meta-Learning-With-Python，在带有注释的 Jupyter Notebook 中查看相应代码。

首先，导入所需的库：

```
import re
import numpy as np
from PIL import Image

from sklearn.model_selection import train_test_split
from keras import backend as K
from keras.layers import Activation
from keras.layers import Input, Lambda, Dense, Dropout, Convolution2D,
MaxPooling2D, Flatten
from keras.models import Sequential, Model
from keras.optimizers import RMSprop
```

现在，定义一个函数来读取输入图像。`read_image` 函数以图像为输入，返回一个 NumPy 数组：

```
def read_image(filename, byteorder='>'):
    #首先将图像以 raw 格式读入缓冲区
    with open(filename, 'rb') as f:
        buffer = f.read()
    #使用 regex 提取图片的头部（header）、宽度（width）、高度（height）以及最大值（maxval）
    header, width, height, maxval = re.search(
        b"(^P5\s(?:\s*#.*[\r\n])*"
        b"(\d+)\s(?:\s*#.*[\r\n])*"
        b"(\d+)\s(?:\s*#.*[\r\n])*"
        b"(\d+)\s(?:\s*#.*[\r\n]\s)*)", buffer).groups()
    #然后，使用 np.frombuffer（该函数用于将缓冲区转换为一维数组）将图像转换为 NumPy 数组
    return np.frombuffer(buffer,
                        dtype='u1' if int(maxval) < 256 else
byteorder+'u2',
                        count=int(width)*int(height),
                        offset=len(header)
                        ).reshape((int(height), int(width)))
```

例如，打开一个图像（见图 2-7）：

```
Image.open("data/orl_faces/s1/1.pgm")
```



图　2-7

当我们将图像输入 `read_image` 函数时，会返回一个 NumPy 数组：

```
img = read_image('data/orl_faces/s1/1.pgm')
img.shape
(112, 92)
```

现在定义另一个函数 `get_data` 来生成数据。正如我们所知，对于孪生网络，数据应该是成对的（正和负），并带有二元标签。

首先从同一个目录中读取 `(img1,img2)` 图像，将它们存储在 `x_genuine_pair` 数组中，并将 `y_genuine` 赋值为 `1`。然后，从不同目录读取 `(img1,img2)` 图像，将它们存储在 `x_imposite` 对中，并将 `y_imposite` 赋值为 `0`。

最后，我们将 `x_genuine_pair` 和 `x_imposite` 拼接（concatenate）成 `X`，将 `y_genuine` 和 `y_imposite` 拼接成 `Y`：

```
size = 2
total_sample_size = 10000


def get_data(size, total_sample_size):
    #读取图像
    image = read_image('data/orl_faces/s' + str(1) + '/' + str(1) + '.pgm','rw+')
    #缩减尺寸
    image = image[::size, ::size]
    #获取新尺寸
    dim1 = image.shape[0]
    dim2 = image.shape[1]

    count = 0
    #使用[total_sample, no_of_pairs, dim1, dim2]的形状初始化NumPy 数组
    x_geuine_pair = np.zeros([total_sample_size, 2, 1, dim1, dim2]) # 参数值2代表一
对样本
    y_genuine = np.zeros([total_sample_size, 1])
    for i in range(40):
        for j in range(int(total_sample_size/40)):
            ind1 = 0
            ind2 = 0
            #从同一个目录中读取图像（正样本对）
            while ind1 == ind2:
                ind1 = np.random.randint(10)
                ind2 = np.random.randint(10)
            #读取两个图像
```

```
            img1 = read_image('data/orl_faces/s' + str(i+1) + '/' +
str(ind1 + 1) + '.pgm', 'rw+')
            img2 = read_image('data/orl_faces/s' + str(i+1) + '/' +
str(ind2 + 1) + '.pgm', 'rw+')
            #缩减尺寸
            img1 = img1[::size, ::size]
            img2 = img2[::size, ::size]
            #将图像存入已初始化的NumPy数组中
            x_geuine_pair[count, 0, 0, :, :] = img1
            x_geuine_pair[count, 1, 0, :, :] = img2
            #因为我们是从同一目录中取出图像的，所以为其分配标签1（正样本对）
            y_genuine[count] = 1
            count += 1

    count = 0
    x_imposite_pair = np.zeros([total_sample_size, 2, 1, dim1, dim2])
    y_imposite = np.zeros([total_sample_size, 1])
    for i in range(int(total_sample_size/10)):
        for j in range(10):
            #从不同的目录中读取图像（负样本对）
            while True:
                ind1 = np.random.randint(40)
                ind2 = np.random.randint(40)
                if ind1 != ind2:
                    break
            img1 = read_image('data/orl_faces/s' + str(ind1+1) + '/' +
str(j + 1) + '.pgm', 'rw+')
            img2 = read_image('data/orl_faces/s' + str(ind2+1) + '/' +
str(j + 1) + '.pgm', 'rw+')

            img1 = img1[::size, ::size]
            img2 = img2[::size, ::size]

            x_imposite_pair[count, 0, 0, :, :] = img1
            x_imposite_pair[count, 1, 0, :, :] = img2
            #因为我们是从不同的目录中取出图像的，所以为其分配标签0（负样本对）
            y_imposite[count] = 0
            count += 1
    #现在，将正样本对与负样本对拼接成完整的数据
    X = np.concatenate([x_geuine_pair, x_imposite_pair], axis=0)/255
    Y = np.concatenate([y_genuine, y_imposite], axis=0)

    return X, Y
```

现在，我们生成数据并检查数据大小。如你所见，我们有20 000个数据点，其中10 000个是正样本对，其余的10 000个是负样本对：

```
X, Y = get_data(size, total_sample_size)

X.shape
(20000, 2, 1, 56, 46)

Y.shape
(20000, 1)
```

接下来，我们将数据划分为 75%的训练数据和 25%的测试数据：

```
x_train, x_test, y_train, y_test = train_test_split(X, Y, test_size=.25)
```

现在我们已成功地生成了数据，下面构建孪生网络。首先，定义基网络，它基本上是一个用于特征提取的卷积网络。我们建立两个均带有 ReLU 激活函数和最大池化（max pooling）的卷积层与一个扁平化层（flat layer）：

```
def build_base_network(input_shape):
    seq = Sequential()
    nb_filter = [6, 12]
    kernel_size = 3
    #卷积层 1
    seq.add(Convolution2D(nb_filter[0], kernel_size, kernel_size,
input_shape=input_shape,
                          border_mode='valid', dim_ordering='th'))
    seq.add(Activation('relu'))
    seq.add(MaxPooling2D(pool_size=(2, 2)))
    seq.add(Dropout(.25))
    #卷积层 2
    seq.add(Convolution2D(nb_filter[1], kernel_size, kernel_size,
border_mode='valid', dim_ordering='th'))
    seq.add(Activation('relu'))
    seq.add(MaxPooling2D(pool_size=(2, 2), dim_ordering='th'))
    seq.add(Dropout(.25))

    #扁平化层
    seq.add(Flatten())
    seq.add(Dense(128, activation='relu'))
    seq.add(Dropout(0.1))
    seq.add(Dense(50, activation='relu'))
    return seq
```

接下来把图像对输入基网络，它将返回嵌入，即特征向量：

```
input_dim = x_train.shape[2:]
img_a = Input(shape=input_dim)
img_b = Input(shape=input_dim)

base_network = build_base_network(input_dim)
feat_vecs_a = base_network(img_a)
feat_vecs_b = base_network(img_b)
```

`feat_vecs_a` 和 `feat_vecs_b` 是图像对的特征向量。接下来把这些特征向量输入能量函数，计算它们之间的距离，并使用欧氏距离作为能量函数：

```
def euclidean_distance(vects):
    x, y = vects
    return K.sqrt(K.sum(K.square(x - y), axis=1, keepdims=True))


def eucl_dist_output_shape(shapes):
    shape1, shape2 = shapes
    return (shape1[0], 1)
```

```
distance = Lambda(euclidean_distance,
output_shape=eucl_dist_output_shape)([feat_vecs_a, feat_vecs_b])
```

现在将轮数设置为 13，并使用 RMS prop 进行优化，定义模型：

```
epochs = 13
rms = RMSprop()

model = Model(input=[input_a, input_b], output=distance)
```

接下来将损失函数定义为 contrastive_loss 函数，并编译模型：

```
def contrastive_loss(y_true, y_pred):
    margin = 1
    return K.mean(y_true * K.square(y_pred) + (1 - y_true) *
K.square(K.maximum(margin - y_pred, 0)))

model.compile(loss=contrastive_loss, optimizer=rms)
```

现在训练模型：

```
img_1 = x_train[:, 0]
img_2 = x_train[:, 1]

model.fit([img_1, img_2], y_train, validation_split=.25, batch_size=128,
verbose=2, nb_epoch=epochs)
```

可以看到，损失是如何随着训练轮数增加而减少的：

```
Train on 11250 samples, validate on 3750 samples
Epoch 1/13
 - 60s - loss: 0.2179 - val_loss: 0.2156
Epoch 2/13
 - 53s - loss: 0.1520 - val_loss: 0.2102
Epoch 3/13
 - 53s - loss: 0.1190 - val_loss: 0.1545
Epoch 4/13
 - 55s - loss: 0.0959 - val_loss: 0.1705
Epoch 5/13
 - 52s - loss: 0.0801 - val_loss: 0.1181
Epoch 6/13
 - 52s - loss: 0.0684 - val_loss: 0.0821
Epoch 7/13
 - 52s - loss: 0.0591 - val_loss: 0.0762
Epoch 8/13
 - 52s - loss: 0.0526 - val_loss: 0.0655
Epoch 9/13
 - 52s - loss: 0.0475 - val_loss: 0.0662
Epoch 10/13
 - 52s - loss: 0.0444 - val_loss: 0.0469
Epoch 11/13
 - 52s - loss: 0.0408 - val_loss: 0.0478
Epoch 12/13
 - 52s - loss: 0.0381 - val_loss: 0.0498
Epoch 13/13
 - 54s - loss: 0.0356 - val_loss: 0.0363
```

现在使用测试数据进行预测：

```
pred = model.predict([x_test[:, 0], x_test[:, 1]])
```

接下来定义一个函数来计算准确率：

```
def compute_accuracy(predictions, labels):
    return labels[predictions.ravel() < 0.5].mean()
```

下面是模型的准确率：

```
compute_accuracy(pred, y_test)
```

```
0.9779092702169625
```

## 2.3　使用孪生网络进行音频识别

在上一节，我们了解了如何使用孪生网络来识别人脸。下面我们来看看如何使用孪生网络来识别音频。我们将训练网络来区分狗和猫的叫声。猫狗音频数据集可以从这里下载：https://www.kaggle.com/mmoreaux/audio-catsand-dogs#cats_dogs.zip。

下载后，将数据分成 3 个文件夹：Dogs、Sub_dogs 和 Cats。在 Dogs 和 Sub_dogs 中存放狗叫声的音频，在 Cats 中存放猫叫声的音频。我们网络的目标是识别出声音是狗叫声还是其他声音。我们知道，对于孪生网络，需要将数据成对输入。我们从 Dogs 和 Sub_dogs 中各选择一个音频并将其标记为一个正样本对，从 Dogs 和 Cats 中各选择一个音频并将其标记为一个负样本对。也就是说，(dogs, subdogs)是正样本对，(dogs, cats)是负样本对。

下面我们一步一步地展示如何训练孪生网络来识别出声音是狗叫声还是其他声音。

为了更好地理解，可以在 GitHub 网站搜索 Hands-On-Meta-Learning-With-Python，在带有注释的 Jupyter Notebook 中查看相应代码。

首先，导入所有需要的库：

```
#导入基本的库
import glob
import IPython
from random import randint

#数据处理
import librosa
import numpy as np

#建模
from sklearn.model_selection import train_test_split

from keras import backend as K
```

```
from keras.layers import Activation
from keras.layers import Input, Lambda, Dense, Dropout, Flatten
from keras.models import Model
from keras.optimizers import RMSprop
```

加载并聆听如下音频片段：

```
IPython.display.Audio("data/audio/Dogs/dog_barking_0.wav")

IPython.display.Audio("data/audio/Cats/cat_13.wav")
```



那么，如何将这些原始音频输入网络中呢？如何从原始音频中提取有意义的特征呢？众所周知，神经网络只接受向量化的输入，因此需要将音频转换成特征向量。该怎么做呢？可以通过几种机制为音频生成嵌入。一个常用的机制是**梅尔频率倒谱系数**（Mel-Frequency Cepstral Coefficients，MFCC）。它使用对数功率谱的线性余弦变换在非线性梅尔频率范围内转换音频的短期功率谱。要了解更多关于 MFCC 的信息，请查看 PRACTICAL CRYPTOGRAPHY 网站上的优秀教程 *Mel Frequency Cepstral Coefficient (MFCC) Tutorial*。

我们将使用 librosa 库中的 MFCC 函数来生成音频嵌入。因此，定义一个名为 `audio2vector` 的函数，它返回给定音频文件的音频嵌入：

```
def audio2vector(file_path, max_pad_len=400):
    #读取音频文件
    audio, sr = librosa.load(file_path, mono=True)

    #缩减形状
    audio = audio[::3]
    #使用 MFCC 提取音频嵌入
    mfcc = librosa.feature.mfcc(audio, sr=sr)
    #由于不同音频的音频嵌入长度各不相同，我们将最大长度设为 400
    #使用 0 填充

    pad_width = max_pad_len - mfcc.shape[1]
    mfcc = np.pad(mfcc, pad_width=((0, 0), (0, pad_width)),
mode='constant')

    return mfcc
```

我们将加载一个音频文件，并查看其嵌入：

```
audio_file = 'data/audio/Dogs/dog_barking_0.wav'
audio2vector(audio_file)
array([[-297.54905127, -288.37618855, -314.92037769,  ...,       0.        ,
           0.        ,    0.        ],
       [  23.05969394,    9.55913148,   37.2173831 ,  ...,       0.        ,
           0.        ,    0.        ],
       [-122.06299523, -115.02627567, -108.18703056,  ...,       0.        ,
           0.        ,    0.        ],
       ...,
       [  -6.40930836,   -2.8602708 ,   -2.12551478,  ...,       0.        ,
```

```
       0.        ,    0.        ],
    [  0.70572914,    4.21777791,    4.62429301,  ...,      0.      ,
       0.        ,    0.        ],
    [ -6.08997702,  -11.40687886,  -18.2415214 ,  ...,      0.      ,
       0.        ,    0.        ]])
```

了解了如何生成音频嵌入后，需要为孪生网络创建数据。正如我们所知，孪生网络接受成对的数据，因此我们定义获取数据的函数。我们将创建正样本对(Dogs, Sub_dogs)和负样本对(Dogs, Cats)，并将标签分别赋值为 1 和 0：

```
def get_data():
    pairs = []
    labels = []
    Dogs = glob.glob('data/audio/Dogs/*.wav')
    Sub_dogs = glob.glob('data/audio/Sub_dogs/*.wav')
    Cats = glob.glob('data/audio/Cats/*.wav')
    np.random.shuffle(Sub_dogs)
    np.random.shuffle(Cats)
    for i in range(min(len(Cats),len(Sub_dogs))):
        #负样本对
        if (i % 2) == 0:
pairs.append([audio2vector(Dogs[randint(0,3)]),audio2vector(Cats[i])])
            labels.append(0)
        #正样本对
        else:
pairs.append([audio2vector(Dogs[randint(0,3)]),audio2vector(Sub_dogs[i])])
            labels.append(1)
    return np.array(pairs), np.array(labels)

X, Y = get_data("/home/sudarshan/sudarshan/Experiments/oneshotaudio/data/")
```

接下来划分数据，将 75%的数据用于训练，25%的数据用于测试：

```
X_train, X_test, y_train, y_test = train_test_split(X, Y, test_size=0.2)
```

我们已成功地生成了数据，下面构建孪生网络。我们定义基网络，用于特征提取，并使用 3 个密集层，中间有一个 dropout 层：

```
def build_base_network(input_shape):
    input = Input(shape=input_shape)
    x = Flatten()(input)
    x = Dense(128, activation='relu')(x)
    x = Dropout(0.1)(x)
    x = Dense(128, activation='relu')(x)
    x = Dropout(0.1)(x)
    x = Dense(128, activation='relu')(x)
    return Model(input, x)
```

接下来将音频对输入基网络，它将返回特征：

```
input_dim = X_train.shape[2:]
audio_a = Input(shape=input_dim)
audio_b = Input(shape=input_dim)
```

```
base_network = build_base_network(input_dim)
feat_vecs_a = base_network(audio_a)
feat_vecs_b = base_network(audio_b)
```

feat_vecs_a 和 feat_vecs_b 是音频对的特征向量。接下来将这些特征向量输入能量函数，计算它们之间的距离，并使用欧氏距离作为能量函数：



```
def euclidean_distance(vects):
    x, y = vects
    return K.sqrt(K.sum(K.square(x - y), axis=1, keepdims=True))


def eucl_dist_output_shape(shapes):
    shape1, shape2 = shapes
    return (shape1[0], 1)

distance = Lambda(euclidean_distance,
output_shape=eucl_dist_output_shape)([feat_vecs_a, feat_vecs_b])
```

接下来，将轮数设置为 13，并使用 RMS prop 进行优化：

```
epochs = 13
rms = RMSprop()

model = Model(input=[audio_a, audio_b], output=distance)
```

最后，将损失函数定义为 contrastive_loss 函数，并编译模型：

```
def contrastive_loss(y_true, y_pred):
    margin = 1
    return K.mean(y_true * K.square(y_pred) + (1 - y_true) *
K.square(K.maximum(margin - y_pred, 0)))

model.compile(loss=contrastive_loss, optimizer=rms)
```

现在训练模型：

```
audio1 = X_train[:, 0]
audio2 = X_train[:, 1]

model.fit([audio_1, audio_2], y_train, validation_split=.25,
          batch_size=128, verbose=2, nb_epoch=epochs)
```

可以看到，损失是如何随着训练轮数增加而减少的：

```
Train on 8 samples, validate on 3 samples
Epoch 1/13
 - 0s - loss: 23594.8965 - val_loss: 1598.8439
Epoch 2/13
 - 0s - loss: 62360.9570 - val_loss: 816.7302
Epoch 3/13
 - 0s - loss: 17967.6230 - val_loss: 970.0378
Epoch 4/13
 - 0s - loss: 20030.3711 - val_loss: 358.9078
```

```
Epoch 5/13
 - 0s - loss: 11196.0547 - val_loss: 339.9991
Epoch 6/13
 - 0s - loss: 3837.2898 - val_loss: 381.9774
Epoch 7/13
 - 0s - loss: 2037.2965 - val_loss: 303.6652
Epoch 8/13
 - 0s - loss: 1434.4321 - val_loss: 229.1388
Epoch 9/13
 - 0s - loss: 2553.0562 - val_loss: 215.1207
Epoch 10/13
 - 0s - loss: 1046.6870 - val_loss: 197.1127
Epoch 11/13
 - 0s - loss: 569.4632 - val_loss: 183.8586
Epoch 12/13
 - 0s - loss: 759.0131 - val_loss: 162.3362
Epoch 13/13
 - 0s - loss: 819.8594 - val_loss: 120.3017
```

## 2.4　小结

在本章，我们学习了什么是孪生网络，以及如何使用孪生网络建立人脸识别模型和音频识别模型；研究了孪生网络的架构，它基本上由两个具有相同权重和架构的神经网络组成，这些网络的输出可以插入某个能量函数中以计算相似性。

在下一章，我们将学习原型网络及其变体，如高斯原型网络和半原型网络，还将学习如何使用原型网络进行 omniglot 字符集分类。

## 2.5　思考题

(1) 什么是孪生网络？
(2) 什么是对比损失函数？
(3) 什么是能量函数？
(4) 孪生网络理想的数据格式是怎样的？
(5) 孪生网络有哪些应用？

## 2.6　延伸阅读

- 用于目标跟踪的孪生网络：参见 Luca Bertinetto、Jack Valmadre、João F. Henriques 等人的文章 *Fully-Convolutional Siamese Networks for Object Tracking*。
- 用于图像识别的孪生网络：参见 Gregory Koch、Richard Zemel 和 Ruslan Salakhutdinov 的文章 *Siamese Neural Networks for One-shot Image Recognition*。

第 3 章

# 原型网络及其变体



在上一章，我们学习了什么是孪生网络，如何使用它们来执行少样本学习任务，以及如何使用它们进行人脸识别和音频识别。在这一章，我们将学习另一个有趣的少样本学习算法——原型网络。它对训练集中不存在的类别也具有泛化能力。我们将理解什么是原型网络，如何使用原型网络在 omniglot 数据集中执行一个分类任务，以及不同的原型网络变体，如高斯原型网络和半原型网络。

本章内容包括：

- 原型网络；
- 原型网络算法；
- 使用原型网络进行分类；
- 高斯原型网络；
- 高斯原型网络算法；
- 半原型网络。

## 3.1 原型网络

原型网络是另一种简单、高效的少样本学习算法。与孪生网络一样，原型网络也试图学习度量空间来进行分类。原型网络的基本思想是创建每个类的原型表示，并根据类原型与查询点之间的距离对查询点（新点）进行分类。

假设我们有一个由狮子、大象和狗的图像组成的支撑集，如图 3-1 所示。

| 图像<br>($x_i$) | 标签<br>($y_i$) |
| --- | --- |
| | 狮子 |
| | 狮子 |
| | 大象 |
| | 大象 |
| | 狗 |
| | 狗 |

支撑集

图　3-1

因此，我们有 3 个类：{狮子, 大象, 狗}。现在需要为这 3 个类中的每个类创建一个原型表示。如何构建这 3 个类的原型呢？首先，我们将使用嵌入函数学习每个数据点的嵌入（如图 3-2 所示）。嵌入函数 $f_\phi()$ 可以是任何能够用来提取特征的函数。由于输入是图像，因此可以使用卷积网络作为嵌入函数，它将从输入图像中提取特征。

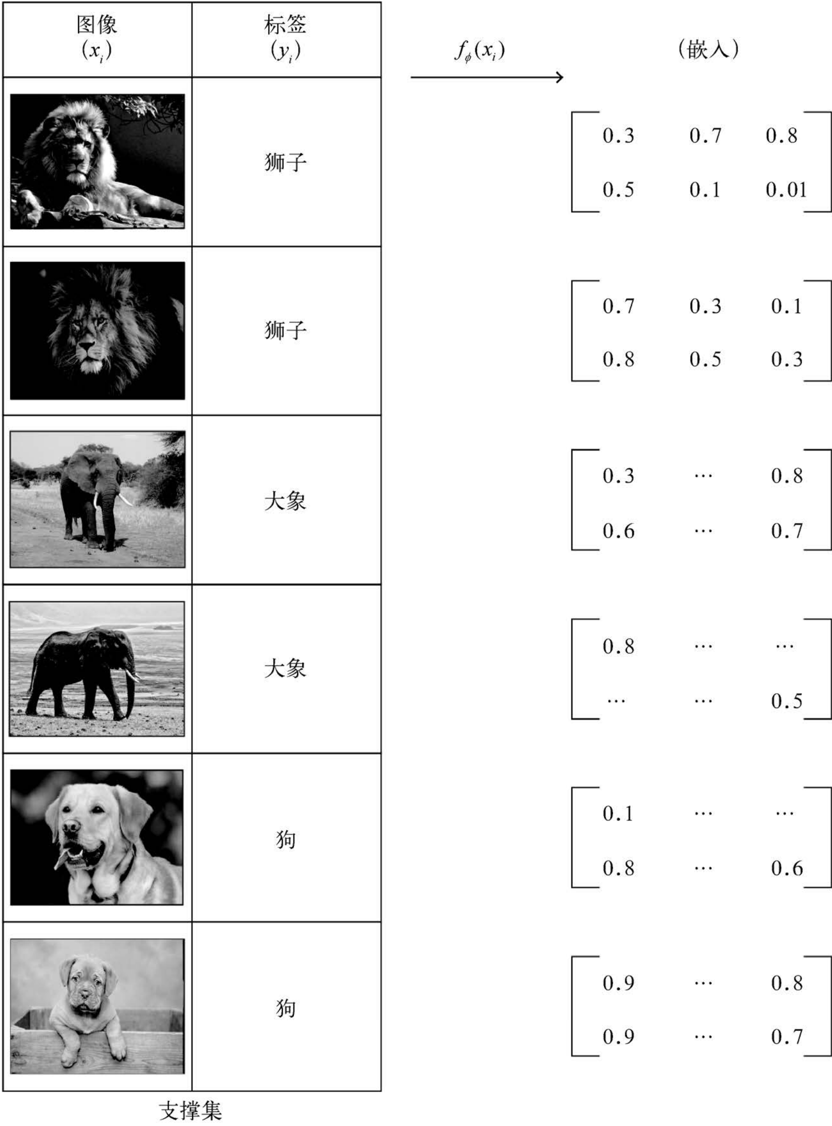



图 3-2

一旦我们学习了每个数据点的嵌入，就可以对每个类中数据点的嵌入取平均数，并形成类原型，如图 3-3 所示。因此，类原型基本上是类中数据点的平均嵌入。

图 3-3

同样，当输入一个新的数据点，即一个查询点（我们将预测它的标签）时，我们将使用与创建类原型相同的嵌入函数，来为这个新数据点生成嵌入。也就是说，我们使用卷积网络为查询点生成嵌入（如图 3-4 所示）。

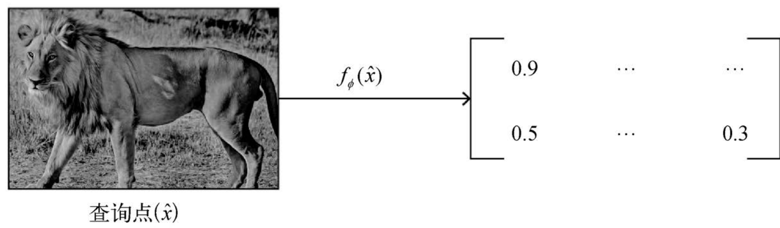



图 3-4

一旦有了查询点的嵌入，就可以比较类原型和查询点嵌入之间的距离，从而确定查询点属于哪个类。可以使用欧氏距离来度量类原型与查询点嵌入之间的距离，如图 3-5 所示。

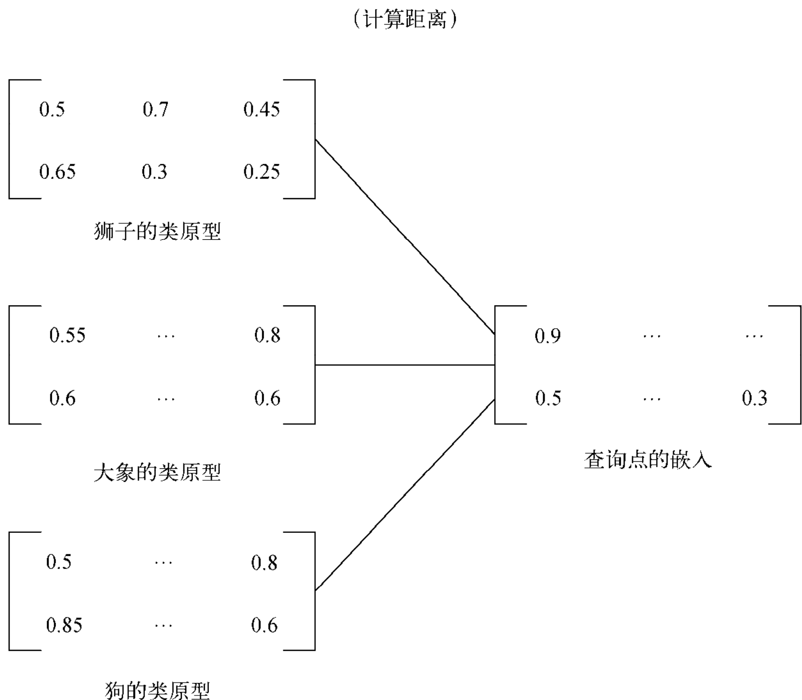



图 3-5

在找到类原型和查询点嵌入之间的距离之后，我们对这个距离应用 softmax 并得到概率。因为我们有 3 个类——狮子、大象和狗，所以会得到 3 个概率。概率最大的类就是查询点的类。

因为我们希望网络只从少量数据点学习，也就是说，希望执行少样本学习，所以用同样的方法训练网络。因此，我们使用阶段式训练——对于每个阶段，随机抽取数据集中每个类的几个数据点作为样本，称之为支撑集，并且只使用支撑集而不是整个数据集来训练网络。类似地，我们从数据集中随机抽取一个点作为查询点，并尝试预测它的类。因此，通过这种方式，我们的网络将学会如何从一组较少的数据点中学习。

原型网络的总体流程如图 3-6 所示。如你所见，首先，我们将为支撑集中的所有数据点生成嵌入，并通过类中数据点的平均嵌入来构建类原型，还为查询点生成嵌入。然后，计算类原型与查询点嵌入之间的距离，用欧氏距离作为距离度量。之后，对这个距离应用 softmax 并得到概率。如图 3-6 所示，因为查询点是狮子，所以狮子的概率很高，为 0.9。

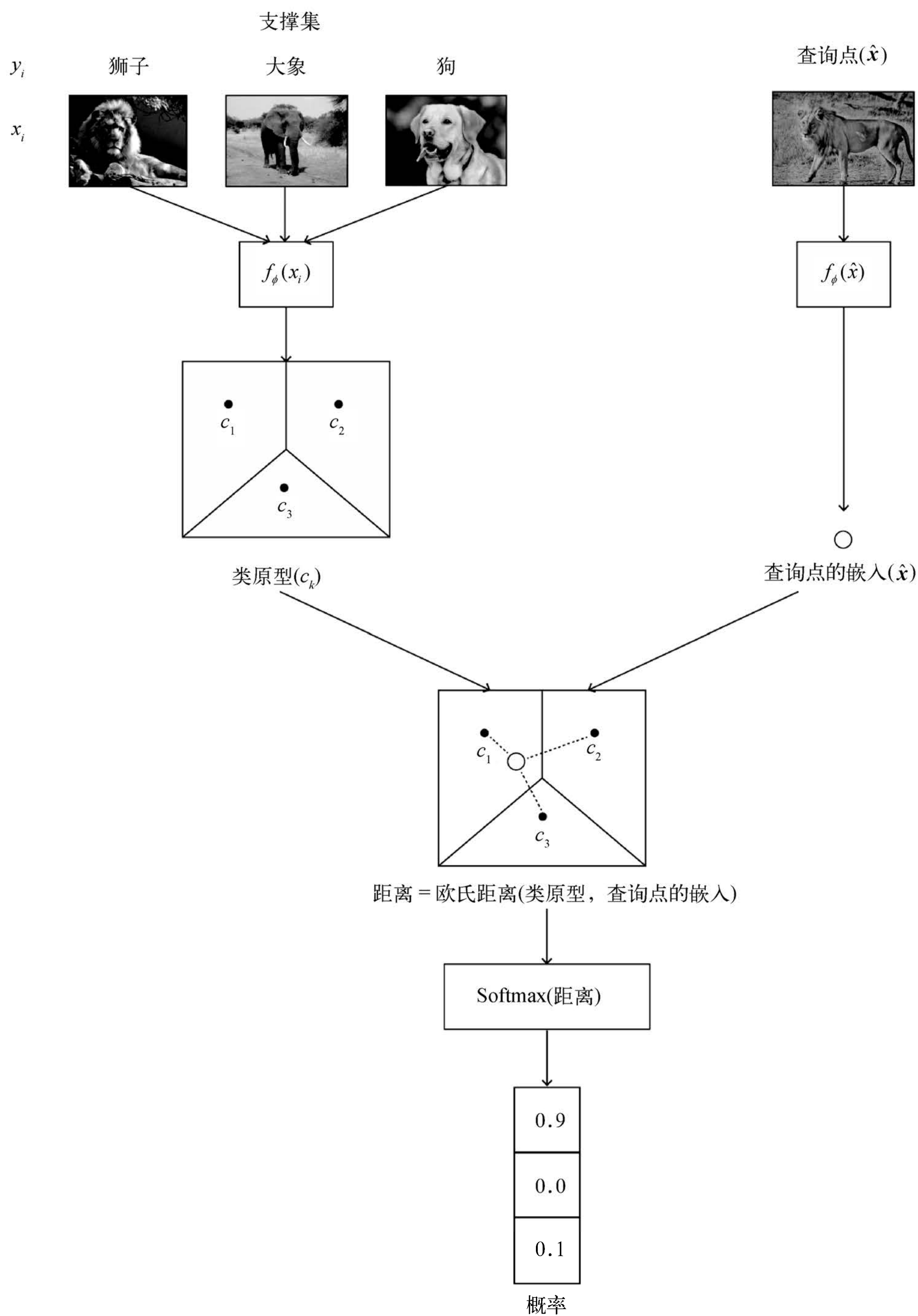

支撑集
$y_i$
狮子
大象
狗
$x_i$
查询点($\hat{x}$)
$f_\phi(x_i)$
$f_\phi(\hat{x})$
$c_1$
$c_2$
$c_3$
类原型($c_k$)
查询点的嵌入($\hat{x}$)
距离 = 欧氏距离(类原型，查询点的嵌入)
Softmax(距离)
0.9
0.0
0.1
概率

图　3-6

原型网络不仅用于单样本/少样本学习，而且还用于零样本学习。考虑这种情况：每个类没有数据点，但元信息包含每个类的高级描述。因此，在这些情况下，我们从每个类的元信息中学习嵌入，以形成类原型，然后使用类原型执行分类。

### 3.1.1 算法

原型网络的算法如下。

(1) 假设有个数据集 $D=\{(x_1,y_1),(x_2,y_2),\cdots,(x_n,y_n)\}$，其中 $x$ 是特征， $y$ 是类标签。

(2) 在阶段性训练中，从数据集 $D$ 的各类别中分别随机抽样 $n$ 个数据点，以组成支撑集 $S$ 。



(3) 同样，选择 $n$ 个数据点组成查询集 $Q$ 。

(4) 使用嵌入函数 $f_\phi()$ 学习支撑集中数据点的嵌入。嵌入函数可以是任何特征提取器，例如用于图像的卷积网络与用于文本的 LSTM 网络。

(5) 一旦有了每个数据点的嵌入，就可以通过求每个类中数据点的平均嵌入，计算出每个类的原型：

$$\text{类原型}(c)=\frac{1}{S}\sum_{(x_i,y_i)\in s} f_\phi(x_i)$$

(6) 同样，学习查询集的嵌入。

(7) 计算查询集嵌入与类原型之间的欧氏距离 $d$ 。

(8) 通过对 $d$ 使用 softmax，预测查询集属于某个类别的概率 $p_\phi(y=k\mid x)$ ：

$$p_\phi(y=k\mid x)=\frac{\exp(-d(f_\phi(x),c))}{\sum_k \exp(-d(f_\phi(x),c))}$$

(9) 计算负对数概率损失函数 $J(\phi)=-\log p_\phi(y=k\mid x)$ ，并使用随机梯度下降法最小化损失。

### 3.1.2 使用原型网络执行分类

本节将介绍如何使用原型网络执行分类任务。我们使用 omniglot 数据集来执行分类。这个数据集包括来自 50 种字母表的 1623 个手写字符，每个字符有 20 个不同的例子，由不同的人书写。因为我们想让网络从数据中学习，所以用同样的方法训练它。从每个类中抽取 5 个示例，并将其用作支撑集。我们使用由 4 个卷积块组成的序列作为编码器，学习支撑集的嵌入，并构建类原型。类似地，我们从每个类中抽取 5 个示例作为查询集，学习查询集嵌入，并通过比较查询集嵌入和类原型之间的欧氏距离来预测查询集的类。下面逐步进行解析。

你也可以在 GitHub 网站搜索 Hands-On-Meta-Learning-With-Python，在带有注释的 Jupyter Notebook 中查看相应代码。

首先，按如下代码导入所有需要的库。

```
import os
import glob
from PIL import Image
import numpy as np
import tensorflow as tf
```

现在要看看数据中有什么。我们知道，我们有来自不同字母表的不同字符，其中每个字符有20种变体，由20个不同的人书写。让我们把其中的一些字母画出来，并进行检查。

日语中的字符（见图3-7）：

```
Image.open('data/images/Japanese_(katakana)/character13/0608_01.png')
```

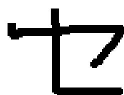


图 3-7

由不同的人书写的同一字符（见图3-8）：

```
Image.open('data/images/Japanese_(katakana)/character13/0608_13.png')
```

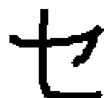


图 3-8

梵文中的字符（见图3-9与图3-10）：

```
Image.open('data/images/Sanskrit/character13/0863_09.png')
```

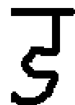


图 3-9

```
Image.open('data/images/Sanskrit/character13/0863_13.png')
```



图 3-10

如何把这些图像转换成一个数组呢？可以使用 np.array 将这些图像转换为一个数组，并将其大小变形为 28×28：

```
image_name = 'data/images/Sanskrit/character13/0863_13.png'
alphabet, character, rotation = 'Sanskrit/character13/rot000'.split('/')
rotation = float(rotation[3:])
```

输出如下：

```
array([[1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.], [1., 1., 1., 1., 1., 1., 1.,
1., 1., 1., 1., 1., 1., 1., 1., 0., 1., 1., 1., 1., 1., 1., 0., 1., 1., 1.,
0., 1.], [1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.], [1., 1., 1., 1., 1., 1.,
1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
1., 1., 1.]],dtype=float32)
```

在知道数据集中有什么之后，我们载入数据集：

```
root_dir = 'data/'
```

更多详细信息请参考/data/omniglot/split/train.txt 文件（包含语言名称、旋转信息与字符数量）和/data/omniglot/data/文件夹（包含图像）：

```
train_split_path = os.path.join(root_dir, 'splits', 'train.txt')

with open(train_split_path, 'r') as train_split:
    train_classes = [line.rstrip() for line in train_split.readlines()]
```

找到类的数量如下：

```
#类的数量
no_of_classes = len(train_classes)
```

按如下代码，将样本量设为 20，因为数据集中的每个类都有 20 个样本。同时，将图像的宽高设为 28 × 28。

```
#样本数量
num_examples = 20

#图像宽度
img_width = 28

#图像高度
img_height = 28
channels = 1
```

接下来初始化训练数据集，其形状为类的数量、样本的数量，以及图像的高度和宽度：

```
train_dataset = np.zeros([no_of_classes, num_examples, img_height,
img_width], dtype=np.float32)
```

现在，读取所有的图像，将它们转换成一个 NumPy 数组，并将其连同标签和值一起存储在

train_dataset 数组中，即 train_dataset = [label, values]：

```
for label, name in enumerate(train_classes):
    alphabet, character, rotation = name.split('/')
    rotation = float(rotation[3:])
    img_dir = os.path.join(root_dir, 'data', alphabet, character)
    img_files = sorted(glob.glob(os.path.join(img_dir, '*.png')))
    for index, img_file in enumerate(img_files):
        values = 1. -
np.array(Image.open(img_file).rotate(rotation).resize((img_width,
img_height)), np.float32, copy=False)
        train_dataset[label, index] = values
```

训练数据的形状如下：

```
train_dataset.shape

(4112, 20, 28, 28)
```

在加载训练数据之后，需要为它们创建嵌入。因为输入值是图像，所以使用卷积运算生成嵌入。因此，定义一个包含 64 个过滤器的卷积块，并使用批量标准化（batch normalization）和 ReLU 为激活函数。然后，执行最大池化操作：

```
def convolution_block(inputs, out_channels, name='conv'):

    conv = tf.layers.conv2d(inputs, out_channels, kernel_size=3,
padding='SAME')
    conv = tf.contrib.layers.batch_norm(conv, updates_collections=None,
decay=0.99, scale=True, center=True)
    conv = tf.nn.relu(conv)
    conv = tf.contrib.layers.max_pool2d(conv, 2)
    return conv
```

现在，定义嵌入函数，它为我们提供包含 4 个卷积块的嵌入：

```
def get_embeddings(support_set, h_dim, z_dim, reuse=False):

        net = convolution_block(support_set, h_dim)
        net = convolution_block(net, h_dim)
        net = convolution_block(net, h_dim)
        net = convolution_block(net, z_dim)
        net = tf.contrib.layers.flatten(net)
        return net
```



记住，不使用整个数据集进行训练。因为我们使用单样本学习，所以从每个类中抽取一些数据点作为支撑集，并阶段性地使用支撑集训练网络。

现在，定义一些重要的变量——考虑一个有 50 个类别、5 个样本的学习场景：

```
#类的数量
num_way = 50
```

```
#支撑集中每个类的样本数量
num_shot = 5

#查询集中查询点的数量
num_query = 5

#样本数量
num_examples = 20

h_dim = 64
z_dim = 64
```

接着，为支撑集和查询集初始化占位符：

```
support_set = tf.placeholder(tf.float32, [None, None, img_height,
img_width, channels])
query_set = tf.placeholder(tf.float32, [None, None, img_height, img_width,
channels])
```

然后，将支撑集和查询集的形状分别存储在 support_set_shape 与 query_set_shape 中。

```
support_set_shape = tf.shape(support_set)
query_set_shape = tf.shape(query_set)
```

我们得到类的数量、支撑集中数据点的数量以及（用于初始化支撑集和查询集的）查询集中数据点的数量：

```
num_classes, num_support_points = support_set_shape[0],
support_set_shape[1]
num_query_points = query_set_shape[1]
```

接下来，为标签定义占位符：

```
y = tf.placeholder(tf.int64, [None, None])

#将标签转换为独热编码
y_one_hot = tf.one_hot(y, depth=num_classes)
```

然后使用嵌入函数，为支撑集生成嵌入：

```
support_set_embeddings = get_embeddings(tf.reshape(support_set,
[num_classes * num_support_points, img_height, img_width, channels]),
h_dim, z_dim)
```

计算每个类的原型，它是类的支撑集嵌入的均值向量：

```
embedding_dimension = tf.shape(support_set_embeddings)[-1]

class_prototype = tf.reduce_mean(tf.reshape(support_set_embeddings,
[num_classes, num_support_points, embedding_dimension]), axis=1)
```

接下来，使用相同的嵌入函数来获得查询集的嵌入：

```
query_set_embeddings = get_embeddings(tf.reshape(query_set, [num_classes *
num_query_points, img_height, img_width, channels]), h_dim, z_dim,
reuse=True)
```

在获得类原型和查询集嵌入后，我们定义一个距离函数，它给出了类原型和查询集嵌入之间的距离：

```
def euclidean_distance(a, b):

    N, D = tf.shape(a)[0], tf.shape(a)[1]
    M = tf.shape(b)[0]
    a = tf.tile(tf.expand_dims(a, axis=1), (1, M, 1))
    b = tf.tile(tf.expand_dims(b, axis=0), (N, 1, 1))
    return tf.reduce_mean(tf.square(a - b), axis=2)
```

计算类原型和查询集嵌入之间的距离：

```
distance = euclidean_distance(class_prototype,query_set_embeddings)
```

接下来，将距离输入 softmax 函数，得到每个类的概率：

```
predicted_probability = tf.reshape(tf.nn.log_softmax(-distance),
[num_classes, num_query_points, -1])
```

然后，计算损失：

```
loss = -tf.reduce_mean(tf.reshape(tf.reduce_sum(tf.multiply(y_one_hot,
predicted_probability), axis=-1), [-1]))
```

准确率计算方式如下：

```
accuracy =
tf.reduce_mean(tf.to_float(tf.equal(tf.argmax(predicted_probability,
axis=-1), y)))
```

然后，使用 Adam 优化器来最小化损失：

```
train = tf.train.AdamOptimizer().minimize(loss)
```

现在，启动 TensorFlow 会话并训练模型：

```
sess = tf.InteractiveSession()
init = tf.global_variables_initializer()
sess.run(init)
```

定义轮数与阶段数：

```
num_epochs = 20
num_episodes = 100
```

接下来开始阶段性训练，也就是说，对于每一个阶段，我们抽样数据点，构建支撑集和查询集，并训练模型：

```
for epoch in range(num_epochs):
    for episode in range(num_episodes):
        #选择60个类
        episodic_classes = np.random.permutation(no_of_classes)[:num_way]
        support = np.zeros([num_way, num_shot, img_height, img_width],
dtype=np.float32)
        query = np.zeros([num_way, num_query, img_height, img_width],
dtype=np.float32)
        for index, class_ in enumerate(episodic_classes):
            selected = np.random.permutation(num_examples)[:num_shot +
num_query]
            support[index] = train_dataset[class_, selected[:num_shot]]
            #每个类5个查询点
            query[index] = train_dataset[class_, selected[num_shot:]]
        support = np.expand_dims(support, axis=-1)
        query = np.expand_dims(query, axis=-1)
        labels = np.tile(np.arange(num_way)[:, np.newaxis], (1,
num_query)).astype(np.uint8)
        _, loss_, accuracy_ = sess.run([train, loss, accuracy],
feed_dict={support_set: support, query_set: query, y:labels})
        if (episode+1) % 20 == 0:
            print('Epoch {} : Episode {} : Loss: {}, Accuracy:
{}'.format(epoch+1, episode+1, loss_, accuracy_))
```

## 3.2 高斯原型网络

下面我们来看一个原型网络的变体，叫作高斯原型网络。我们刚刚了解了原型网络是如何学习数据点的嵌入、如何通过获取每个类的平均嵌入来构建类原型，并使用类原型执行分类的。

在高斯原型网络中，除了生成数据点的嵌入外，我们还在数据点周围添加了一个以高斯协方差矩阵为特征的置信区域（confidence region）[①]。置信区域有助于描述单个数据点的质量，在数据有噪声的、同质性低的情况下十分有用。

因此，在高斯原型网络中，编码器的输出是嵌入以及协方差矩阵。我们没有使用完整的协方差矩阵，而是包括了协方差矩阵的一个半径分量（radius component）或对角分量（diagonal component）以及嵌入。

- **半径分量**：如果使用协方差矩阵的半径分量，那么协方差矩阵的维数就是1，因为半径只是一个数字。
- **对角分量**：如果使用协方差矩阵的对角分量，那么协方差矩阵的维数会与嵌入矩阵维数相同。

同样，我们用协方差矩阵的逆矩阵，而不直接使用协方差矩阵。可以使用以下任何一种方法将原始协方差矩阵转换为逆协方差矩阵。设 $\boldsymbol{S}_{\text{raw}}$ 为协方差矩阵，$\boldsymbol{S}$ 为逆协方差矩阵：

① 置信区域是 confidence region，多维；置信区间是 confidence interval，一维。——译者注

- $\boldsymbol{S} = 1 + \text{softplus}(\boldsymbol{S}_{\text{raw}})$
- $\boldsymbol{S} = 1 + \text{sigmoid}(\boldsymbol{S}_{\text{raw}})$
- $\boldsymbol{S} = 1 + 4 \times \text{sigmoid}(\boldsymbol{S}_{\text{raw}})$
- $\boldsymbol{S} = \text{offset} + \text{scale} \times \text{softplus}(\boldsymbol{S}_{\text{raw}} / \text{div})$，其中 offset 与 scale 是可训练参数（trainable parameter）



因此，编码器除了为输入生成嵌入外，还返回协方差矩阵。我们使用协方差矩阵的对角分量或半径分量，还使用逆协方差矩阵（inverse covariance matrix），而不直接使用协方差矩阵。

那么协方差矩阵和嵌入有什么用呢？如前所述，它拓宽了数据点周围的置信区域，这在有噪声的数据中非常有用。请看图 3-11。假设有两个类，A 和 B。黑点表示数据点的嵌入，黑点周围的圆圈表示协方差矩阵，大的虚线圆表示一个类的总体协方差矩阵，中间的星号表示类原型。正如你所看到的，嵌入周围的协方差矩阵给了我们一个围绕数据点和类原型的置信区域。

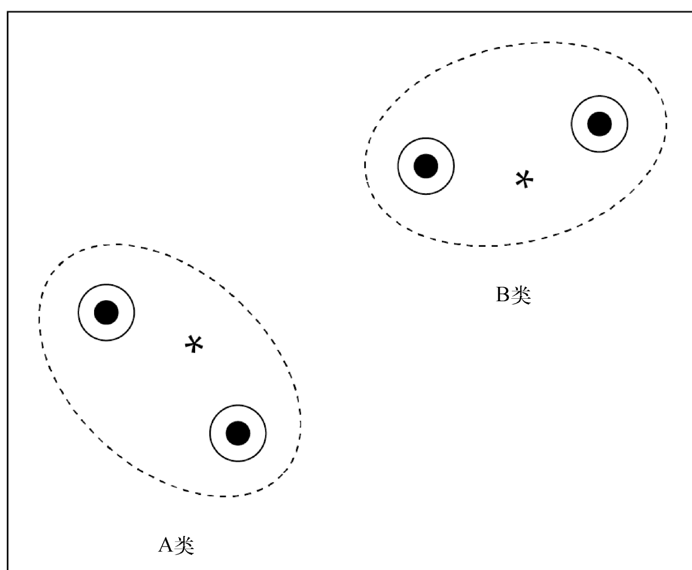



图　3-11

让我们通过阅读代码来更好地理解这一点。假设有一个图像 x，我们想生成这个图像的嵌入。我们用 sigma 表示协方差矩阵。首先，选择要使用协方差矩阵的哪个分量，也就是说，要使用对角分量还是半径分量。如果用半径分量，那么协方差矩阵的维数就是 1；如果选择对角分量，则协方差矩阵的大小会与嵌入维数相同：

```
if component =='radius':
    covariance_matrix_dim = 1
else:
    covariance_matrix_dim = embedding_dim
```

下面定义编码器。因为输入是一个图像，所以使用卷积块作为编码器。我们定义过滤器的大小、数量和池化层（pooling layer）大小：

```
filters = [3,3,3,3]
num_filters = [64,64,64,embedding_dim +covariance_matrix_dim]
pools = [2,2,2,2]
```

初始化图像嵌入 X：

```
previous_channels = 1
embeddings = X
weight = []
bias = []
conv_relu = []
conv = []
conv_pooled = []
```

然后，执行卷积操作，并获取嵌入：

```
for i in range(len(filters)):

    filter_size = filters[i]
    num_filter = num_filters[i]
    pool = pools[i]
    weight.append(tf.get_variable("weights_"+str(i), shape=[filter_size,
filter_size, previous_channels, num_filter])
    bias.append(tf.get_variable("bias_"+str(i), shape=[num_filter]))
    conv.append(tf.nn.conv2d(embeddings, weight[i], strides=[1,1,1,1],
padding='SAME') + bias[i])
    conv_relu.append(tf.nn.relu(conv[i]))
    conv_pooled.append(tf.nn.max_pool(conv_relu[i], ksize =
[1,pool,pool,1], strides=[1,pool,pool,1], padding = "VALID"))

    previous_channels = num_filter
    embeddings = conv_pooled [i]
```

将最后一个卷积层的输出作为嵌入，对结果进行变形，得到嵌入和协方差矩阵：

```
X_encoded = tf.reshape(embeddings,[-1,embedding_dim + covariance_matrix_dim])
```

下面拆分嵌入和原始协方差矩阵（如下代码所示），因为我们需要将原始协方差矩阵转换为逆协方差矩阵。

```
embeddings, raw_covariance_matrix = tf.split(X_encoded, [embedding_dim,
covariance_matrix_dim], 1)
```

接下来，使用任意讨论过的方法计算协方差矩阵的逆：

```
if inverse_transform_type == "softplus":
    offset = 1.0
    scale = 1.0
    inv_covariance_matrix = offset + scale *
```

```
tf.nn.softplus(raw_covariance_matrix)


elif inverse_transform_type == "sigmoid":
    offset = 1.0
    scale = 1.0
    inv_covariance_matrix = offset + scale *
tf.sigmoid(raw_covariance_matrix)


elif inverse_transform_type == "sigmoid_2":
    offset = 1.0
    scale = 4.0
    inv_covariance_matrix = offset + scale *
tf.sigmoid(raw_covariance_matrix)

elif inverse_transform_type == "other":

    init = tf.constant(1.0)
    scale = tf.get_variable("scale", initializer=init)
    div = tf.get_variable("div", initializer=init)
    offset = tf.get_variable("offset", initializer=init)

    inv_covariance_matrix = offset + scale *
tf.nn.softplus(raw_covariance_matrix/div)
```

目前为止，我们已经计算了协方差矩阵和一个输入的嵌入。接下来怎么做？如何计算类原型？类原型 $\overrightarrow{p_c}$ 计算如下：

$$\overrightarrow{p_c} = \frac{\sum_i s_i^c \cdot x_i^c}{\sum_i s_i^c}$$

其中，$s_i^c$ 是逆协方差矩阵的对角分量，$x_i^c$ 是嵌入，上标 $c$ 代表类。

在计算了每个类的原型后，我们学习查询点的嵌入。设 $\overrightarrow{x'}$ 为查询点的嵌入，然后计算查询点嵌入到类原型之间的距离：

$$\text{距离} = \sqrt{(\overrightarrow{x'} - \overrightarrow{p_c})^{\mathrm{T}} \vec{s}_i^c \cdot (\overrightarrow{x'} - \overrightarrow{p_c})}$$

最后，预测查询集 $\hat{y}$ 的类，它与类原型的距离最小：

$$\hat{y} = \operatorname{argmin}_c \text{（距离）}$$

## 算法

现在，我们逐步来理解高斯原型网络。

(1) 假设有一个数据集 $D = \{(x_1, y_1), (x_2, y_2), \cdots, (x_i, y_i)\}$，其中 $x$ 是特征， $y$ 是类标签。假设有

一个二元标签，这意味着只有两个类，`0` 和 `1`。我们将从数据集 $D$ 的各类别中不放回地随机抽样数据点，以组成支撑集 $S$ 。

(2) 同样，随机抽样数据点组成查询集 $Q$ 。

(3) 将支撑集传入嵌入函数 $f()$ 。嵌入函数将生成支撑集的嵌入以及协方差矩阵。

(4) 计算协方差矩阵的逆。

(5) 计算支撑集中每个类的原型，如下：

$$\text{原型}\ (\overrightarrow{p_c}) = \frac{\sum_i \vec{s}_i^c \cdot \vec{x}_i^c}{\sum_i \vec{s}_i^c}$$

其中，$s_i^c$ 是逆协方差矩阵的对角分量，$x_i^c$ 是支撑集的嵌入，上标 $c$ 代表类。

(6) 在计算了支撑集中每个类的原型后，我们学习查询集的嵌入 $Q$ 。设 $x'$ 是查询点的嵌入。

(7) 计算查询点嵌入与类原型之间的距离，如下：

$$\text{距离} = \sqrt{(\overrightarrow{x'} - \overrightarrow{p_c})^T \vec{s}_i^c \cdot (\overrightarrow{x'} - \overrightarrow{p_c})}$$

(8) 计算了类原型与查询集嵌入之间的距离后，预测查询集 $\hat{y}$ 的类，它与类原型的距离最小：

$$\hat{y} = \text{argmin}_c\ (\text{距离})$$

## 3.3 半原型网络

本节我们学习另一种有趣的原型网络变体——半原型网络。它处理未标记的样本。我们知道，在原型网络中，通过每个类的平均嵌入来计算每个类的原型，然后通过计算查询点到类原型的距离来预测查询集的类。

如果数据集包含一些未标记的数据点呢？该如何计算这些未标记数据点的类原型？

假设有支撑集 $S = (x_1, y_1), (x_2, y_2), \cdots, (x_k, y_k)$ ，以及查询集 $Q = (x_1', y_1'), (x_2', y_2'), \cdots, (x_k', y_k')$ ，其中 $x$ 是特征，$y$ 是标签。除了这些，还有一个未标记集合 $R = \{\tilde{x}_1, \tilde{x}_2, \cdots, \tilde{x}_k\}$ 。

可以用这个未标记集合做些什么呢？

首先，我们将使用支撑集中的所有样本来计算类原型。接下来，使用 soft $k$-means 并为 $R$ 中未标记的样本指派类别，即通过计算类原型和未标记的样本之间的欧氏距离，来为 $R$ 中未标记的样本指派类别。

然而，这种方法的问题是，由于我们使用的是 soft $k$-means，所有未标记的样本可能属于任何类原型。比如说，支撑集中有 3 个类{狮子, 大象, 狗}，如果未标记的样本有一个数据点代表一只猫，那么把它放在支撑集的任何一类中都是无意义的。因此，为未标记的样本指派一个新类，

称为干扰类（distractor class），而不是将数据点添加到现有类。

但是，即使使用这种方法，我们也会遇到另一个问题，因为干扰类本身具有高方差。例如，考虑未标记集合 $R$，它包含完全不相关的数据点，例如{猫, 直升机, 公交车, 其他}。在本例中，不建议将所有未标记的样本都保存在单个名为干扰类的类中，因为它们已是受污染的数据了，且彼此无关。

因此，我们将干扰类更改为所有不在类原型的某个阈值距离范围内的样本。该如何计算这个阈值呢？首先，计算未标记集合 $R$ 中的未标记样本到所有类原型之间的归一化距离。接下来，通过将归一化距离的各种统计量（如最小值、最大值、偏度和峰度）输入神经网络，计算每个类原型的阈值。基于这个阈值，我们向类原型添加样本或忽略未标记的样本。

## 3.4 小结

在本章，我们学习了原型网络，了解了原型网络如何使用嵌入函数计算类原型，并通过比较类原型和查询集嵌入之间的欧氏距离来预测查询集的类标签。在此基础上，我们使用一个原型网络对 omniglot 数据集进行分类；然后学习了高斯原型网络，它在使用嵌入的同时，使用协方差矩阵来计算类原型；之后研究了用于处理半监督类的半原型网络。下一章将介绍关系网络与匹配网络。

## 3.5 思考题

(1) 什么是原型网络？
(2) 计算嵌入有什么作用？
(3) 如何计算类原型？
(4) 什么是高斯原型网络？
(5) 高斯原型网络与普通原型网络有何区别？
(6) 高斯原型网络中使用了协方差矩阵的哪些不同分量？

## 3.6 延伸阅读

- 原型网络：参见 Jake Snell、Kevin Swersky 和 Richard S. Zemel 的文章 *Prototypical Networks for Few-shot Learning*。
- 高斯原型网络：参见 Stanislav Fort 的文章 *Gaussian Prototypical Networks for Few-Shot Learning on Omniglot*。
- 半原型网络：参见 Mengye Ren、Eleni Triantafillou、Sachin Ravi 等人的文章 *Meta-Learning for Semi-Supervised Few-Shot Classification*。

第 4 章

# 使用 TensorFlow 构建关系网络与匹配网络



在上一章，我们学习了原型网络，以及如何使用原型网络的变体（如高斯原型网络和半原型网络）进行单样本学习，了解了原型网络如何利用嵌入来执行分类任务。

在本章，我们将学习关系网络和匹配网络。首先，我们将了解什么是关系网络，以及在单样本、少样本和零样本学习场景中如何使用关系网络；然后学习如何使用 TensorFlow 构建关系网络；之后学习匹配网络以及它们在少样本学习中的应用，以及用于匹配网络中的不同类型的嵌入函数；最后学习如何在 Tensorflow 中构建匹配网络。

本章内容包括：

- 关系网络；
- 单样本、少样本与零样本学习场景中的关系网络；
- 使用 TensorFlow 构建关系网络；
- 匹配网络；
- 匹配网络的嵌入函数；
- 匹配网络的架构；
- TensorFlow 中的匹配网络。

## 4.1 关系网络

本节我们将学习另一个有趣的单样本学习算法——关系网络。它是最简单、最有效的单样本学习算法之一。我们将探讨如何在单样本、少样本与零样本学习场景中使用关系网络。

### 4.1.1 单样本学习中的关系网络

关系网络由两个重要的函数组成：嵌入函数 $f_{\varphi}$ 和关系函数 $g_{\phi}$。嵌入函数用于从输入中提取

特征。如果输入是图像，那么可以使用卷积网络作为嵌入函数，它会提供图像的特征向量/嵌入。如果输入是文本，那么可以使用 LSTM 网络来获得文本的嵌入。

我们知道，在单样本学习中，每个类只有一个样本。例如，假设支撑集包含 3 个类，每个类有一个示例。如图 4-1 所示，我们有一个包含 3 个类的支撑集{狮子, 大象, 狗}。

| 图像（$x_i$） | 标签（$y_i$） |
| --- | --- |
| | 狮子 |
| | 大象 |
| | 狗 |

图 4-1　支撑集

假设有查询图像 $x_j$，如图 4-2 所示，我们想预测这个查询图像的类。



图 4-2　查询图像($x_j$)

首先，将支撑集中的每个图像 $x_i$ 传递给嵌入函数 $f_{\varphi}(x_i)$，以提取特征。由于支撑集有图像，因此可以使用卷积网络作为嵌入函数来学习嵌入。嵌入函数将给出支撑集中每个数据点的特征向量。同样，我们将通过把查询图像 $x_j$ 传递给嵌入函数 $f_{\varphi}(x_j)$ 来学习该查询图像的嵌入。

因此，一旦有了支撑集的特征向量 $f_\varphi(x_i)$ 和查询集的特征向量 $f_\varphi(x_j)$，我们就使用运算符 $Z$ 将它们组合起来。这里 $Z$ 可以是任意组合算子。我们使用拼接（concatenation）作为运算符来组合支撑集的特征向量和查询集的特征向量，即 $Z(f_\varphi(x_i), f_\varphi(x_j))$。

如图 4-3 所示，下面组合支撑集的特征向量 $f_\varphi(x_i)$ 和查询集的特征向量 $f_\varphi(x_j)$，但这样组合有什么用呢？这有助于我们理解支撑集中图像的特征向量与查询图像的特征向量之间的关系。在上述例子中，它将帮助我们理解狮子、大象和狗的图像特征向量与查询图像的特征向量之间的关系。

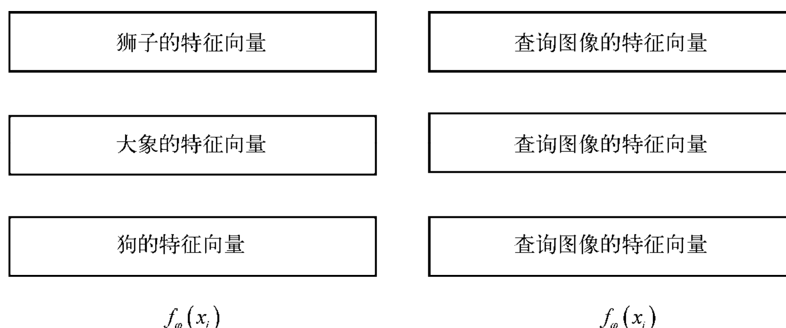



图 4-3 特征拼接

但如何衡量这种关系呢？这就是使用关系函数 $g_\phi$ 的原因。将这些组合的特征向量传递给关系函数，关系函数会生成 0~1 的关系得分，表示支撑集中样本 $x_i$ 与查询集中样本 $x_j$ 之间的相似性。

下面的等式展示了如何计算关系网络中的关系得分：

$$r_{ij} = g_\phi(Z(f_\varphi(x_i), f_\varphi(x_j)))$$

上式中，$r_{ij}$ 表示支撑集中每个类与查询图像之间相似性的关系得分。因为在支撑集中有 3 个类，在查询集中有 1 个图像，所以我们会有 3 个分数来表示支撑集中 3 个类与查询图像的相似性。

关系网络在单样本学习环境中的整体表示如图 4-4 所示。

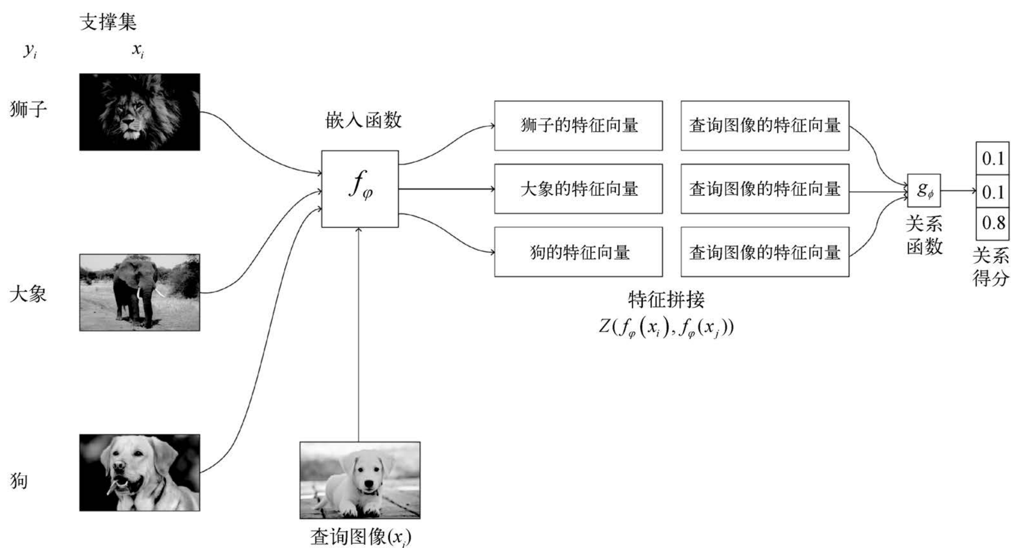



图　4-4

## 4.1.2　少样本学习中的关系网络

我们已了解了如何将属于支撑集中每个类的单个图像与关系网络的单样本学习场景中的查询集中的图像进行比较。但是，在少样本的学习环境中，每个类会有不止一个数据点。如何使用嵌入函数学习特征表示呢？

假设有一个支撑集，其中每个类包含多个图像，如图 4-5 所示。

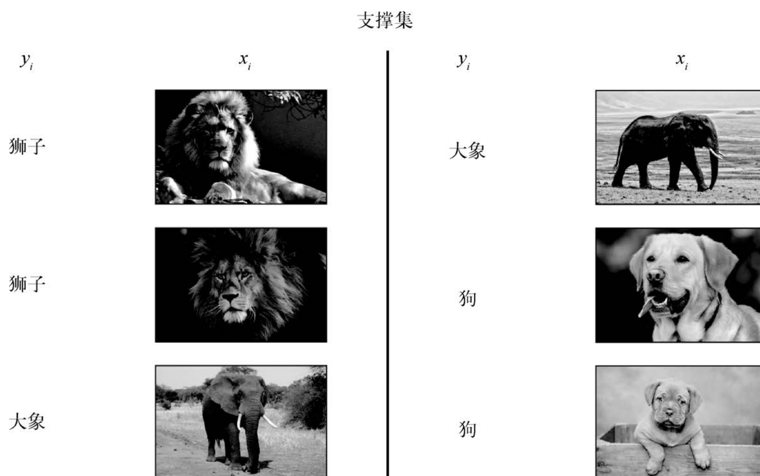



图　4-5

这种情况下，我们将学习支撑集中每个点的嵌入，并对属于每个类的所有数据点的嵌入逐元素相加。因此，我们会得到每个类的嵌入，这是该类中所有数据点的嵌入之和，如图 4-6 所示。

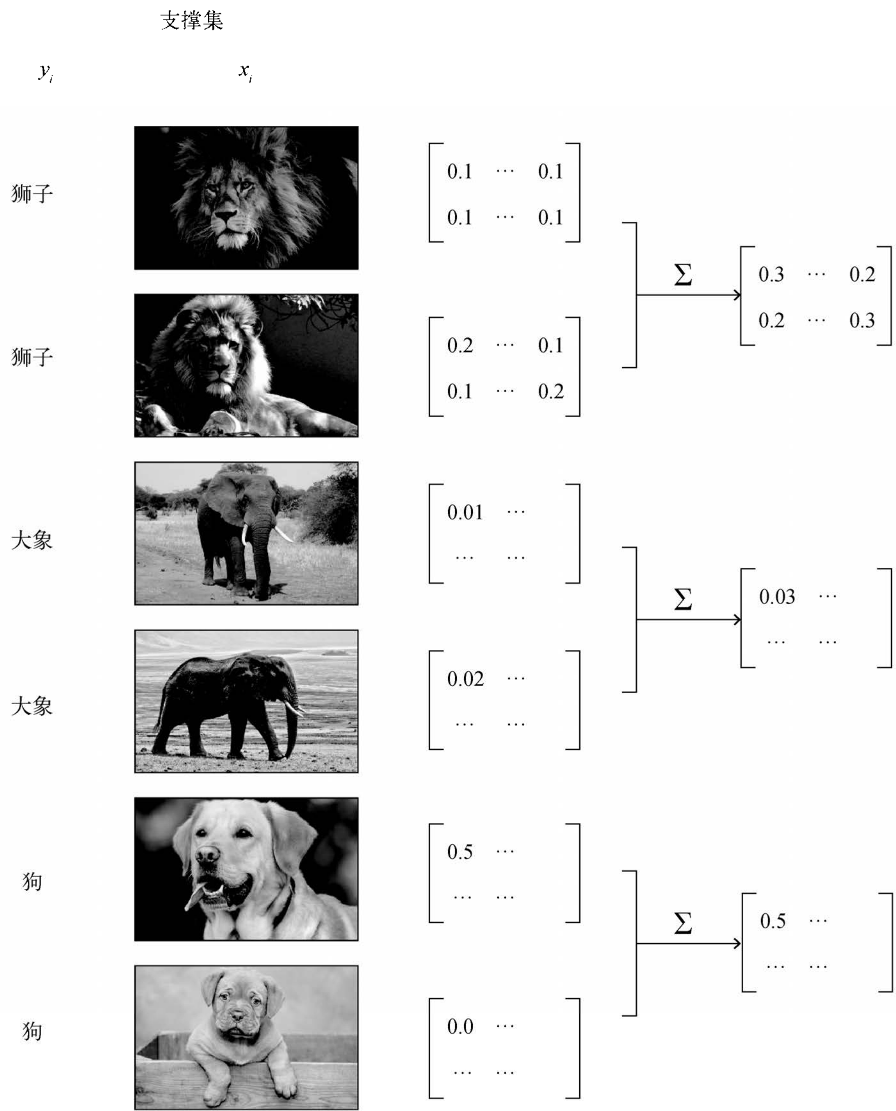



图 4-6

可以像往常一样使用嵌入函数提取查询图像的特征向量。接下来，使用拼接运算符 $Z$ 来结合支撑集的特征向量与查询集的特征向量。我们进行拼接，然后将拼接后的特征向量提供给关系函数并得到关系得分，关系得分表示支撑集中的每个类与查询集中的每个类之间的相似性。

关系网络在少样本学习环境下的整体表示如图 4-7 所示。

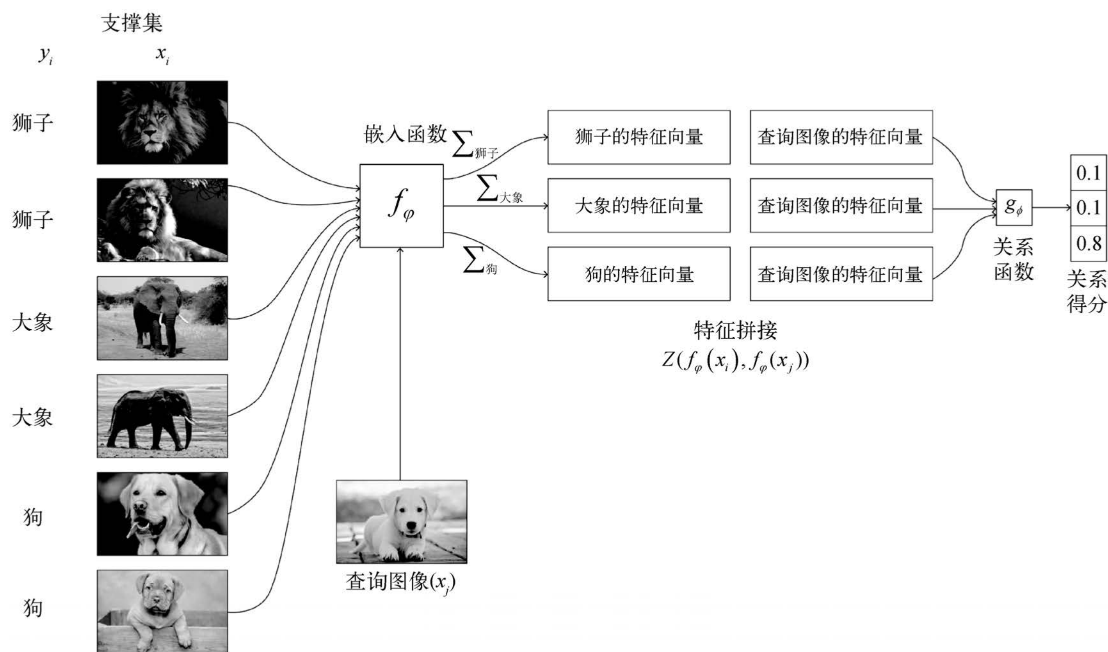



图　4-7

### 4.1.3　零样本学习中的关系网络

现在我们已了解了如何在单样本学习任务和少学习任务中使用关系网络。我们将看到如何在零样本学习场景中使用关系网络。在零样本学习场景中，在每个类下都没有任何数据点，但会有元信息。元信息是关于每个类的属性的信息，元信息会被编码到语义向量 $\boldsymbol{v}_c$ 中，下标 $c$ 表示类别。

我们没有使用单一的嵌入函数来学习支撑集和查询集的嵌入，而是分别使用了两个不同的嵌入函数 $f_{\varphi1}$ 与 $f_{\varphi2}$。首先，我们将使用 $f_{\varphi1}$ 学习语义向量 $\boldsymbol{v}_c$ 的嵌入，并使用 $f_{\varphi2}$ 学习查询集 $x_j$ 的嵌入。下面使用拼接运算符 $Z$ 来拼接这些嵌入：

$$Z(f_{\varphi1}(\boldsymbol{v}_c), f_{\varphi2}(x_j))$$

然后，将结果输入关系函数，并计算关系得分：

$$r_{ij} = g_\phi(Z(f_{\varphi1}(\boldsymbol{v}_c), f_{\varphi2}(x_j)))$$

### 4.1.4 损失函数

关系网络的损失函数是什么？我们将使用均方误差（MSE）作为损失函数。虽然这是一个分类问题，且 MSE 不是分类问题的标准度量，但是关系网络的作者认为，因为预测的是关系得分，所以可以把它当作回归问题。尽管如此，实际上，我们只能自动生成{0,1}目标。

损失函数如下：

$$\varphi,\phi < -\operatorname{argmin}_{\phi,\varphi}\sum_{i=1}^{m}\sum_{j=1}^{n}(r_{i,j}-1(y_i==y_j))^2$$

其中，$\varphi$ 与 $\phi$ 分别是嵌入函数 $f$ 与关系函数 $g$ 的参数。

## 4.2 使用 TensorFlow 构建关系网络



关系函数很简单，对吧？通过在 TensorFlow 中实现一个关系网络，我们将更好地理解关系网络。

你也可以在 GitHub 网站搜索 Hands-On-Meta-Learning-With-Python，在带有注释的 Jupyter Notebook 中查看相应代码。

首先，导入所有需要的库：

```
import tensorflow as tf
import numpy as np
```

然后，随机生成数据点。假设数据集中有两个类，我们将随机为每个类生成大约 1000 个数据点：

```
classA = np.random.rand(1000,18)
ClassB = np.random.rand(1000,18)
```

通过组合这些类来创建数据集：

```
data = np.vstack([classA, ClassB])
```

现在设置标签。将 `classA` 标记为 `1`，将 `classB` 标记为 `0`：

```
label = np.vstack([np.ones((len(classA),1)),np.zeros((len(ClassB),1))])
```

于是，数据集会拥有 2000 条记录：

```
data.shape
(2000, 18)
```

现在为支撑集与查询集定义占位符：

```
xi = tf.placeholder(tf.float32, [None, 9])
xj = tf.placeholder(tf.float32, [None, 9])
```

为 y 标签定义占位符：

```
y = tf.placeholder(tf.float32, [None, 1])
```

下面定义嵌入函数，该函数将学习支撑集和查询集的嵌入。我们将使用一个普通的前馈网络（feedforward network）作为嵌入函数：

```
def embedding_function(x):
    weights = tf.Variable(tf.truncated_normal([9,1]))
    bias = tf.Variable(tf.truncated_normal([1]))
    a = (tf.nn.xw_plus_b(x,weights,bias))
    embeddings = tf.nn.relu(a)
    return embeddings
```

计算支撑集的嵌入：

```
f_xi = embedding_function(xi)
```

计算查询集的嵌入：

```
f_xj = embedding_function(xj)
```

现在已计算了嵌入，得到了特征向量，下面将支撑集和查询集特征向量结合起来：

```
Z = tf.concat([f_xi,f_xj],axis=1)
```

将关系函数定义为具有 ReLU 激活的三层神经网络：

```
def relation_function(x):
    w1 = tf.Variable(tf.truncated_normal([2,3]))
    b1 = tf.Variable(tf.truncated_normal([3]))
    w2 = tf.Variable(tf.truncated_normal([3,5]))
    b2 = tf.Variable(tf.truncated_normal([5]))
    w3 = tf.Variable(tf.truncated_normal([5,1]))
    b3 = tf.Variable(tf.truncated_normal([1]))
    #层 1
    z1 = (tf.nn.xw_plus_b(x,w1,b1))
    a1 = tf.nn.relu(z1)
    #层 2
    z2 = tf.nn.xw_plus_b(a1,w2,b2)
    a2 = tf.nn.relu(z2)
    #层 3
    z3 = tf.nn.xw_plus_b(z2,w3,b3)

    #输出
    y = tf.nn.sigmoid(z3)
    return y
```

现在将支撑集和查询集的特征向量拼接后传递给关系函数，得到关系得分：

```
relation_scores = relation_function(Z)
```

我们将 MSE 作为 loss_function，即 relation_scores 与实际 y 值之间的 squared_difference：

```
loss_function = tf.reduce_mean(tf.squared_difference(relation_scores,y))
```

可以使用 AdamOptimizer 来最小化损失：

```
optimizer = tf.train.AdamOptimizer(0.1)
train = optimizer.minimize(loss_function)
```

现在，启动 TensorFlow 会话：

```
sess = tf.InteractiveSession()
sess.run(tf.global_variables_initializer())
```

现在，随机抽取支撑集 xi 和查询集 xj 的数据点，训练网络：

```
for episode in range(1000):
    _, loss_value = sess.run([train, loss_function],
feed_dict={xi:data[:,0:9]+np.random.randn(*np.shape(data[:,0:9]))*0.05,
xj:data[:,9:]+np.random.randn(*np.shape(data[:,9:]))*0.05,
                                        y:label})
    if episode % 100 == 0:
        print("Episode {}: loss {:.3f} ".format(episode, loss_value))
```

4

输出如下：

```
Episode 0: loss 0.495
Episode 100: loss 0.250
Episode 200: loss 0.250
Episode 300: loss 0.250
Episode 400: loss 0.250
Episode 500: loss 0.250
Episode 600: loss 0.250
Episode 700: loss 0.250
Episode 800: loss 0.250
Episode 900: loss 0.250
```

## 4.3 匹配网络

匹配网络是谷歌 DeepMind 团队发布的另一种简单高效的单样本学习算法。它甚至可以为数据集中未观察到的类生成标签。

假设有一个支撑集 $S$，包含 $K$ 个样本 $(x_1,y_1),(x_2,y_2),\cdots,(x_k,y_k)$。当给定查询点 $\hat{x}$（新的不可见示例）时，匹配网络通过将其与支撑集进行比较来预测 $\hat{x}$ 的类。

我们可以将其定义为 $p(\hat{y}\,|\,\hat{x},S)$，其中 $p$ 为参数化神经网络，$\hat{y}$ 为查询点 $\hat{x}$ 的预测类，$S$ 为支撑集。$p(\hat{y}\,|\,\hat{x},S)$ 会返回 $\hat{x}$ 属于数据集中每个类的概率。然后，选择概率最大的类作为 $\hat{x}$ 的类，但它究竟是如何运作的？该如何计算这个概率？现在让我们来看看。

查询点 $\hat{x}$ 的输出 $\hat{y}$ 的预测方法如下：

$$\hat{y} = \sum_{i=1}^{k} a(\hat{x}, x_i) y_i$$

让我们解析这个方程式。其中，$x_i$ 与 $y_i$ 分别是支撑集的输入和标签。$\hat{x}$ 是查询输入，我们想要预测它的标签。$a$ 是 $\hat{x}$ 与 $x_i$ 之间的注意力机制（attention mechanism），但如何执行注意力呢？这里使用一个简单的注意力机制，即 $\hat{x}$ 与 $x_i$ 之间余弦距离的 softmax 值，即 $a(\hat{x}, x_i) = \mathrm{softmax}(\cos(\hat{x}, x_i))$。

我们不能直接计算原始输入 $\hat{x}$ 与 $x_i$ 之间的余弦距离，因此首先学习它们的嵌入并计算嵌入之间的余弦距离。使用两种不同的嵌入 $f$ 与 $g$，分别用于学习查询输入 $\hat{x}$ 和支撑集输入 $x_i$ 的嵌入。下面的章节会介绍这两个嵌入函数 $f$ 与 $g$，及其嵌入。

因此，可以将注意力方程改写为如下：

$$a(\hat{x}, x_i) = \mathrm{softmax}(\cos(f(\hat{x}), g(x_i)))$$

可以将之前的方程改写为如下：

$$a(\hat{x}, x_i) = \frac{\mathrm{e}^{\cos(f(\hat{x}), g(x_i))}}{\sum_{j=1}^{k} \mathrm{e}^{\cos(f(\hat{x}), g(x_i))}}$$

因此，在计算了注意力矩阵 $a(\hat{x}, x_i)$ 后，用支撑集标签 $y_i$ 乘以注意力矩阵，但如何将支撑集标签 $y_i$ 与注意力矩阵相乘呢？首先，将支撑集标签转换成独热编码值，然后与注意力矩阵相乘，结果，得到 $\hat{y}$ 属于支撑集中每个类的概率。之后，应用 argmax 并选择概率值最大的作为 $\hat{y}$。

如果还不明白的话，请看图 4-8。如你所见，支撑集中有 3 个类{狮子, 大象, 狗}。同时，我们有一个新的查询图像 $\hat{x}$。首先，将支撑集输入嵌入函数 $g$，将查询图像输入嵌入函数 $f$，学习它们的嵌入并计算它们之间的余弦距离。其次，在这个余弦距离上应用 softmax 注意力。再次，使用独热编码的支撑集标签乘以注意力矩阵，得到概率。最后，选择概率最高的那个作为 $\hat{y}$。如图 4-8 所示，查询集图像为大象，在索引 1 处有很高的概率，因此预测 $\hat{y}$ 的类为 1（大象）。

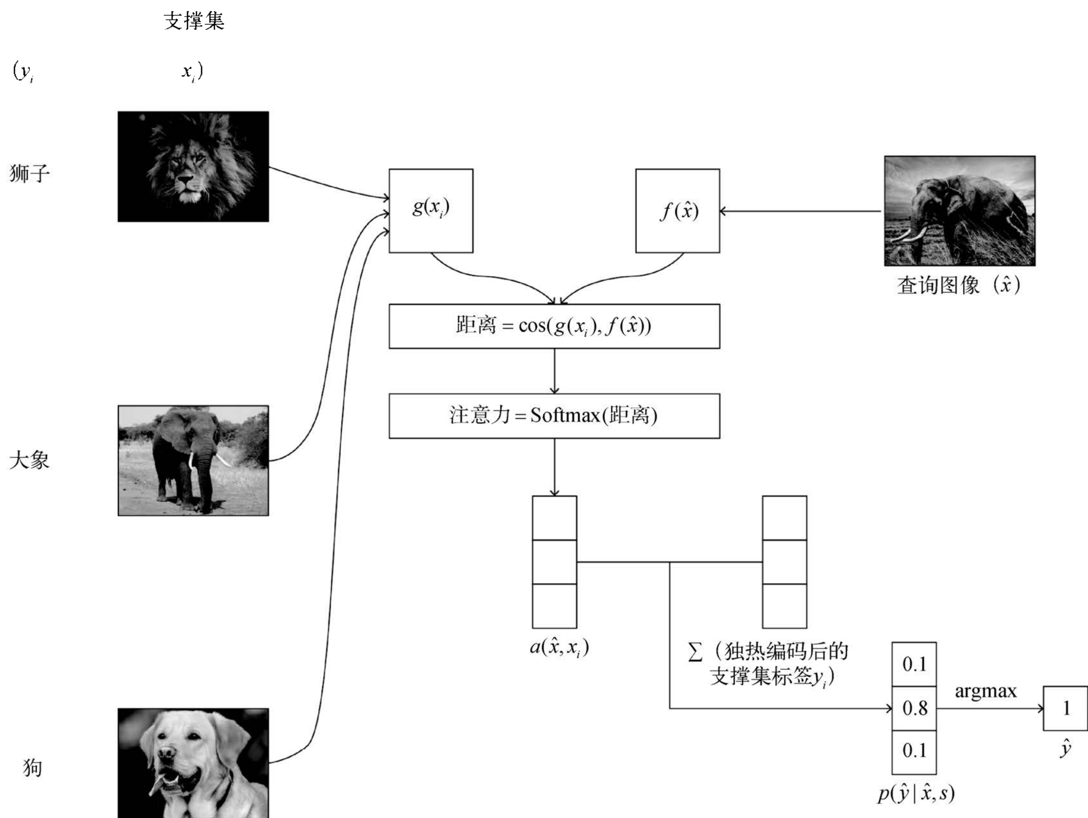



图 4-8

## 嵌入函数

我们已掌握了使用两个嵌入函数 $f$ 与 $g$ 分别学习 $\hat{x}$ 和 $x_i$ 的嵌入，下面来看这两个函数是如何学习嵌入的。

### 1. 支撑集的嵌入函数（$g$）

我们使用嵌入函数 $g$ 来学习支撑集的嵌入，使用双向 LSTM 作为嵌入函数 $g$ 。

嵌入函数 $g$ 的定义如下：

```
def g(X):
    #正向细胞
    forward_cell = rnn.BasicLSTMCell(32)

    #反向细胞
    backward_cell = rnn.BasicLSTMCell(32)
```

4

```
    #双向 LSTM
    outputs, state_forward, state_backward =
rnn.static_bidirectional_rnn(forward_cell, backward_cell, X,
dtype=tf.float32)

    return tf.add(tf.stack(X), tf.stack(outputs))
```

### 2. 查询集的嵌入函数（*f*）

我们使用嵌入函数 $f$ 来学习查询点 $\hat{x}$ 的嵌入，使用 LSTM 作为编码函数。除了使用 $\hat{x}$ 作为输入，我们还将传入支撑集的嵌入，即 $g(x)$，并传递另一个参数 `K`，该参数定义了处理步骤的数量。下面是计算查询集嵌入的步骤。

首先，初始化 LSTM 细胞：

```
cell = rnn.BasicLSTMCell(64)
previous_state = cell.zero_state(batch_size, tf.float32)
```

然后，执行 $K$ 次（处理步骤的数量）如下操作：

```
for step in xrange(K):
```

通过将 $\hat{x}$ 输入 LSTM 细胞，计算查询集的嵌入：

```
output, state = cell(XHat, previous_state)
h_k = tf.add(output, XHat)
```

现在，对支撑集嵌入（即 `g_embedings`）执行 softmax 注意力。它有助于避免不必要的元素：

```
    content_based_attention = tf.nn.softmax(tf.multiply(previous_state[1],
g_embedding))
    r_k = tf.reduce_sum(tf.multiply(content_based_attention, g_embedding),
axis=0)
```

更新 `previous_state` 并重复 $K$ 次这些步骤：

```
previous_state = rnn.LSTMStateTuple(state[0], tf.add(h_k, r_k))
```

计算 `f_embeddings` 的完整代码如下：

```
def f(XHat, g_embedding, K):

    cell = rnn.BasicLSTMCell(64)
    previous_state = cell.zero_state(batch_size, tf.float32)

    for step in xrange(K):
        output, state = cell(XHat, previous_state)
        h_k = tf.add(output, XHat)
        #softmax 注意力
        content_based_attention =
tf.nn.softmax(tf.multiply(previous_state[1], g_embedding))
```

```
        r_k = tf.reduce_sum(tf.multiply(content_based_attention,
g_embedding), axis=0)

        previous_state = rnn.LSTMStateTuple(state[0], tf.add(h_k, r_k))

    return output
```

## 4.4 匹配网络的架构

匹配网络的总体流程如图 4-9 所示，它与我们已见过的图像不同。注意支撑集 $x_i$ 和查询集 $\hat{x}$ 是如何分别通过嵌入函数 $g$ 和 $f$ 计算的。可以看到，嵌入函数 $f$ 将查询集和支撑集的嵌入作为输入。

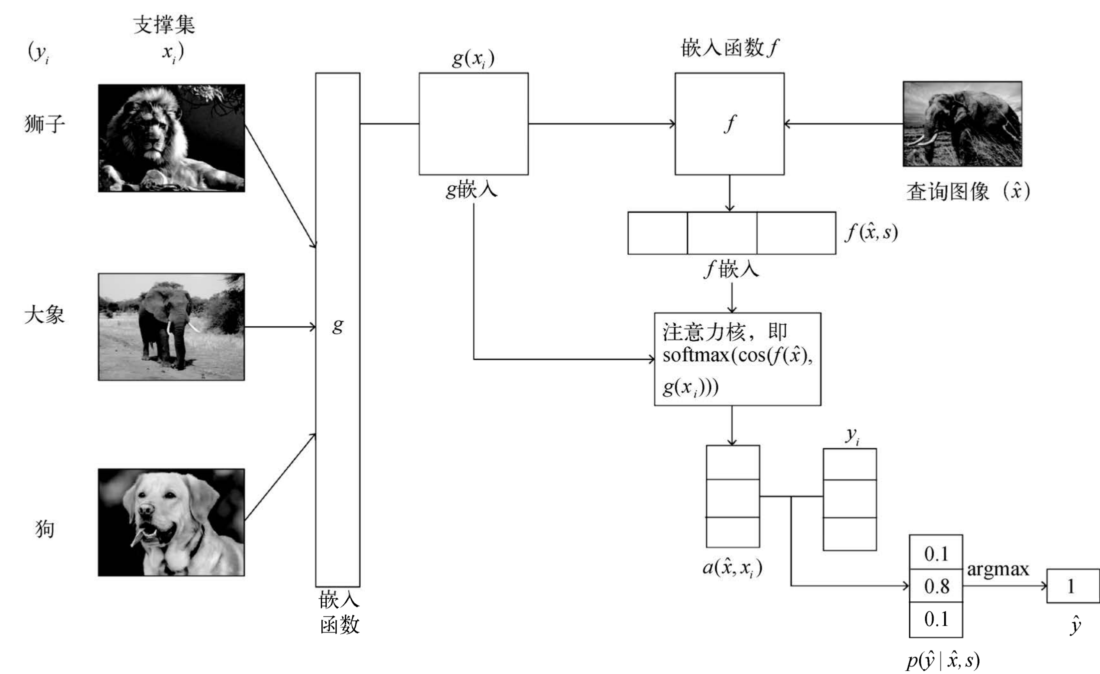



图 4-9

## 4.5 TensorFlow 中的匹配网络

我们现在将逐步了解如何在 TensorFlow 中构建匹配网络。完整代码在最后。

首先，导入所需库：

```
import tensorflow as tf
slim = tf.contrib.slim
rnn = tf.contrib.rnn
```

现在，定义一个 `Matching_network` 类，用于定义网络：

```
class Matching_network():
```

定义__init__方法，用于初始化所有变量：

```
    def __init__(self, lr, n_way, k_shot, batch_size=32):
        #支撑集的占位符
        self.support_set_image = tf.placeholder(tf.float32, [None, n_way *
k_shot, 28, 28, 1])
        self.support_set_label = tf.placeholder(tf.int32, [None, n_way *
k_shot, ])
        #查询集的占位符
        self.query_image = tf.placeholder(tf.float32, [None, 28, 28, 1])
        self.query_label = tf.placeholder(tf.int32, [None, ])
```

假设支撑集和查询集有图像。在将该原始图像输入嵌入函数之前，我们首先使用卷积网络从图像中提取特征，然后将提取的支撑集和查询集的特征分别输入 $g$ 和 $f$ 的嵌入函数。

因此，我们会定义一个称为 `image_encoder` 的函数，用于对图像中的特征进行编码。使用具有最大池化操作的四层卷积网络作为图像编码器：

```
    def image_encoder(self, image):
        with slim.arg_scope([slim.conv2d], num_outputs=64, kernel_size=3,
normalizer_fn=slim.batch_norm):
            #卷积层 1
            net = slim.conv2d(image)
            net = slim.max_pool2d(net, [2, 2])
            #卷积层 2
            net = slim.conv2d(net)
            net = slim.max_pool2d(net, [2, 2])
            #卷积层 3
            net = slim.conv2d(net)
            net = slim.max_pool2d(net, [2, 2])
            #卷积层 4
            net = slim.conv2d(net)
            net = slim.max_pool2d(net, [2, 2])
        return tf.reshape(net, [-1, 1 * 1 * 64])
```

现在定义嵌入函数。我们已在“嵌入函数”一节中看到嵌入函数 $f$ 和 $g$ 是如何定义的。因此，可以直接定义它们：

```
#提取支撑集嵌入的嵌入函数
    def g(self, x_i):

        forward_cell = rnn.BasicLSTMCell(32)
        backward_cell = rnn.BasicLSTMCell(32)
        outputs, state_forward, state_backward =
rnn.static_bidirectional_rnn(forward_cell, backward_cell, x_i,
```

```
dtype=tf.float32)

        return tf.add(tf.stack(x_i), tf.stack(outputs))

    #提取查询集嵌入的嵌入函数
    def f(self, XHat, g_embedding):
        cell = rnn.BasicLSTMCell(64)
        prev_state = cell.zero_state(self.batch_size, tf.float32)

        for step in xrange(self.processing_steps):
            output, state = cell(XHat, prev_state)
            h_k = tf.add(output, XHat)

            content_based_attention =
tf.nn.softmax(tf.multiply(prev_state[1], g_embedding))
            r_k = tf.reduce_sum(tf.multiply(content_based_attention,
g_embedding), axis=0)

            prev_state = rnn.LSTMStateTuple(state[0], tf.add(h_k, r_k))

        return output
```

4

现在，定义函数 `cosine_similarity` 来学习支撑集和查询集嵌入之间的余弦相似度：

```
    def cosine_similarity(self, target, support_set):
        target_normed = target
        sup_similarity = []
        for i in tf.unstack(support_set):
            i_normed = tf.nn.l2_normalize(i, 1)
            similarity = tf.matmul(tf.expand_dims(target_normed, 1),
tf.expand_dims(i_normed, 2))
            sup_similarity.append(similarity)

        return tf.squeeze(tf.stack(sup_similarity, axis=1))
```

最后，使用函数 `train` 来执行训练操作。让我们一步一步来：

```
def train(self, support_set_image, support_set_label, query_image):
```

首先，使用图像编码器编码支撑集图像的特征：

```
     support_set_image_encoded = [self.image_encoder(i) for i in
tf.unstack(support_set_image, axis=1)]
```

然后，使用图像编码器编码查询集图像的特征：

```
query_image_encoded = self.image_encoder(query_image)
```

接下来，使用嵌入函数 $g$ 学习支撑集的嵌入：

```
g_embedding = self.g(support_set_image_encoded)
```

同样，使用嵌入函数 $f$ 来学习查询集的嵌入：

```
f_embedding = self.f(query_image_encoded, g_embedding)
```

现在，计算这两个嵌入之间的 cosine_similarity：

```
    embeddings_similarity = self.cosine_similarity(f_embedding,
g_embedding)
```

然后，对余弦相似度执行 softmax 注意力：

```
attention = tf.nn.softmax(embeddings_similarity)
```

通过将注意力矩阵与独热编码的支撑集标签相乘来预测查询集标签：

```
    y_hat = tf.matmul(tf.expand_dims(attention, 1),
tf.one_hot(support_set_label, self.n_way))
```

之后，得到 probabilities：

```
probabilities = tf.squeeze(y_hat)
```

选择概率最高的索引作为查询图像的类：

```
predictions = tf.argmax(self.logits, 1)
```

最后，定义损失函数。使用 softmax 交叉熵作为损失函数：

```
    loss_function = tf.losses.sparse_softmax_cross_entropy(label,
self.probabilities)
```

使用 AdamOptimizer 来最小化损失函数：

```
tf.train.AdamOptimizer(self.lr).minimize(self.loss_op)
```

现在，可以看到匹配网络完整的最终代码：

```
class Matching_network():
    #初始化所有变量
    def __init__(self, lr, n_way, k_shot, batch_size=32):
        #支撑集的占位符
        self.support_set_image = tf.placeholder(tf.float32, [None, n_way *
k_shot, 28, 28, 1])
        self.support_set_label = tf.placeholder(tf.int32, [None, n_way *
k_shot, ])
        #查询集的占位符
        self.query_image = tf.placeholder(tf.float32, [None, 28, 28, 1])
        self.query_label = tf.placeholder(tf.int32, [None, ])
    #从图像中提取特征的编码函数
    def image_encoder(self, image):
        with slim.arg_scope([slim.conv2d], num_outputs=64, kernel_size=3,
normalizer_fn=slim.batch_norm):
            #卷积层 1
            net = slim.conv2d(image)
            net = slim.max_pool2d(net, [2, 2])
            #卷积层 2
            net = slim.conv2d(net)
            net = slim.max_pool2d(net, [2, 2])
            #卷积层 3
```

```
            net = slim.conv2d(net)
            net = slim.max_pool2d(net, [2, 2])
            #卷积层 4
            net = slim.conv2d(net)
            net = slim.max_pool2d(net, [2, 2])
        return tf.reshape(net, [-1, 1 * 1 * 64])
    #提取支撑集嵌入的嵌入函数
    def g(self, x_i):

        forward_cell = rnn.BasicLSTMCell(32)
        backward_cell = rnn.BasicLSTMCell(32)
        outputs, state_forward, state_backward =
rnn.static_bidirectional_rnn(forward_cell, backward_cell, x_i,
dtype=tf.float32)
        return tf.add(tf.stack(x_i), tf.stack(outputs))

    #提取查询集嵌入的嵌入函数
    def f(self, XHat, g_embedding):
        cell = rnn.BasicLSTMCell(64)
        prev_state = cell.zero_state(self.batch_size, tf.float32)

        for step in xrange(self.processing_steps):
            output, state = cell(XHat, prev_state)
            h_k = tf.add(output, XHat)

            content_based_attention =
tf.nn.softmax(tf.multiply(prev_state[1], g_embedding))
            r_k = tf.reduce_sum(tf.multiply(content_based_attention,
g_embedding), axis=0)

            prev_state = rnn.LSTMStateTuple(state[0], tf.add(h_k, r_k))

        return output

    #余弦相似度函数，用于计算支撑集嵌入与查询集嵌入之间的余弦相似度
    def cosine_similarity(self, target, support_set):
        target_normed = target
        sup_similarity = []
        for i in tf.unstack(support_set):
            i_normed = tf.nn.l2_normalize(i, 1)
            similarity = tf.matmul(tf.expand_dims(target_normed, 1),
tf.expand_dims(i_normed, 2))
            sup_similarity.append(similarity)

        return tf.squeeze(tf.stack(sup_similarity, axis=1))

    def train(self, support_set_image, support_set_label, query_image):
        #使用图像编码器来编码查询集图像的特征
        query_image_encoded = self.image_encoder(query_image)
        #使用图像编码器来编码支撑集图像的特征
        support_set_image_encoded = [self.image_encoder(i) for i in
tf.unstack(support_set_image, axis=1)]
        #使用嵌入函数 g 生成支撑集嵌入
        g_embedding = self.g(support_set_image_encoded)
        #使用嵌入函数 f 生成查询集嵌入
        f_embedding = self.f(query_image_encoded, g_embedding)
```

```
        #计算两种嵌入间的余弦相似度
        embeddings_similarity = self.cosine_similarity(f_embedding,
g_embedding)
        #对余弦相似度执行注意力
        attention = tf.nn.softmax(embeddings_similarity)
        #通过将注意力矩阵与独热编码的支撑集标签相乘来预测查询集标签
        y_hat = tf.matmul(tf.expand_dims(attention, 1),
tf.one_hot(support_set_label, self.n_way))
        #获取概率
        probabilities = tf.squeeze(y_hat)
        #选择概率最高的索引作为查询图像的类
        predictions = tf.argmax(self.probabilities, 1)
        #使用 softmax 交叉熵作为损失函数
        loss_function = tf.losses.sparse_softmax_cross_entropy(label,
self.probabilities)
        #使用 adam 优化器来最小化损失函数
        tf.train.AdamOptimizer(self.lr).minimize(self.loss_op)
```

## 4.6　小结

在本章，我们学习了匹配网络和关系网络在少样本学习中的应用；了解了关系网络如何学习支撑集和查询集的嵌入，并将嵌入组合起来，将它们输入关系函数来计算关系得分；研究了匹配网络如何使用两个不同的嵌入函数来学习支撑集和查询集的嵌入，以及它如何预测查询集的类。

在下一章，我们将通过向存储器中存取信息来学习神经图灵机与记忆增强神经网络的运作方式。

## 4.7　思考题

(1) 关系网络中使用了哪几种函数？
(2) 关系网络中的运算符 $Z$ 是什么？
(3) 什么是关系函数？
(4) 关系网络的损失函数是什么？
(5) 匹配网络中使用了哪几种嵌入函数？
(6) 匹配网络如何预测查询点的类别？

## 4.8　延伸阅读

- 匹配网络：参见 Oriol Vinyals、Charles Blundell、Timothy Lillicrap 等人的文章 *Matching Networks for One Shot Learning*。
- 关系网络：参见 Flood Sung、Yongxin Yang、Li Zhang 等人的文章 *Learning to Compare: Relation Network for Few-Shot Learning*。

第 5 章

# 记忆增强神经网络

5

前几章介绍了几种基于距离的度量学习算法。我们从孪生网络开始，了解了孪生网络如何学会区分两个输入；然后研究了原型网络和原型网络的变体，如高斯原型网络和半原型网络；之后探索了有趣的匹配网络和关系网络。

在这一章，我们将学习记忆增强神经网络（memory-augmented neural networks，MANN），该网络用于单样本学习。在深入研究 MANN 之前，我们将了解它的前身，神经图灵机（Neural Turing Machines，NTM），并学习 NTM 如何利用外存储器存储和检索信息，以及如何使用 NTM 执行复制任务。



本章内容包括：

- NTM；
- NTM 中的读写；
- 寻址机制；
- 使用 NTM 执行复制任务；
- MANN；
- MANN 中的读写。

## 5.1 NTM

NTM 是一种有趣的算法，它能够在存储器中存储和检索信息，其思想是用外存储器来增强神经网络，也就是说，NTM 不是使用隐藏状态作为存储器，而是使用外存储器来存储和检索信息，其架构如图 5-1 所示。

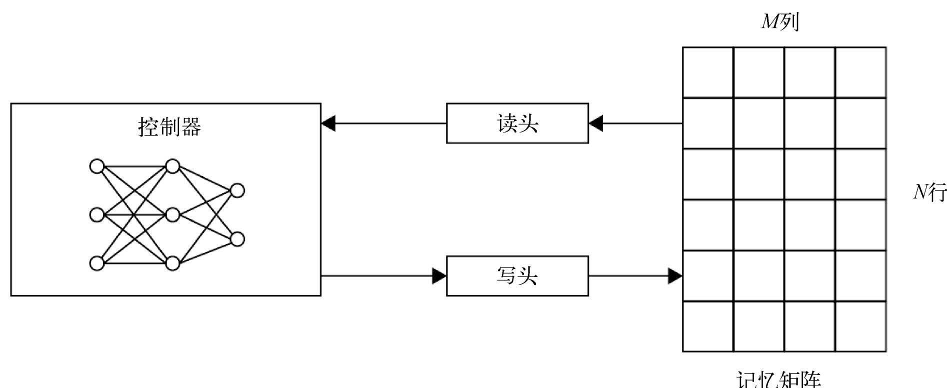



图　5-1

下面是 NTM 的重要组件。

- **控制器**：基本上是前馈神经网络或递归神经网络；对存储器进行读写。
- **存储器**：存储矩阵（memory matrix）、存储体（memory bank）或简单的存储器，是我们存储信息的地方。存储矩阵基本上是由记忆细胞组成的二维矩阵，包含 $N$ 行和 $M$ 列。我们使用控制器从存储器中访问内容。因此，控制器接收来自外部环境的输入，并通过与存储矩阵交互发出响应。
- **读头和写头**：读头和写头是包含存储器地址的指针，它必须从存储器中读写。

但如何访问存储器中的信息呢？可以通过指定行索引和列索引来访问存储器中的信息吗？是的，可以，不过问题是，如果通过索引访问信息，就不能使用梯度下降来训练 NTM，这是因为无法计算索引的梯度。因此，NTM 的作者定义了使用控制器读写的模糊操作（blurry operation）。模糊操作会在一定程度上与存储器中的所有元素交互。它基本上是一种注意力机制，强烈关注存储器中重要的读/写位置，而忽略其他位置。因此，我们使用特殊的读和写操作来确定要关注存储器中的哪个位置。下一节会进一步探讨读写操作。

## 5.1.1　NTM 中的读与写

现在我们来看看如何读写存储矩阵。

### 1. 读操作

读操作从存储器中读取一个值，但存储矩阵中有很多内存块（memory block），我们需要在存储器中选择哪个来读取呢？这由权向量决定。权向量确定存储器中区域的重要性。我们用注意力机制来得到权向量。下一节将详细讨论如何精确地计算权向量。权向量是归一化的，这意味着它的取值范围为 0~1，并且值的和等于 1。图 5-2 为长度 $N$ 的权向量。

| 0.3 | 0.1 | 0.3 | 0.1 | 0.2 |
|---|---|---|---|---|

图　5-2　权向量（$\boldsymbol{w}_t$）

我们用 $\boldsymbol{w}_t$ 表示这个归一化的权向量，其中下标 $t$ 表示时间，$\boldsymbol{w}_t(i)$ 表示权向量中索引 $i$ 、时间 $t$ 处的元素：

$$\sum_i \boldsymbol{w}_t(i)=1,\quad 0\leqslant \boldsymbol{w}_t(i)\leqslant 1,\quad \forall i$$

如图 5-3 所示，存储矩阵由 $N$ 行和 $M$ 列组成。将 $t$ 时刻的存储矩阵记为 $\boldsymbol{M}_t$ 。

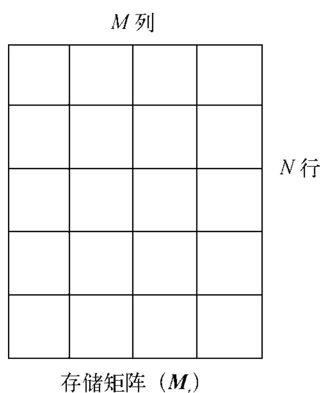



图　5-3

一旦有了权向量和存储矩阵，我们就将存储矩阵 $\boldsymbol{M}_t$ 和权向量 $\boldsymbol{w}_t$ 进行线性组合，得到读向量（read vector）$\boldsymbol{r}_t$ ，如图 5-4 所示。

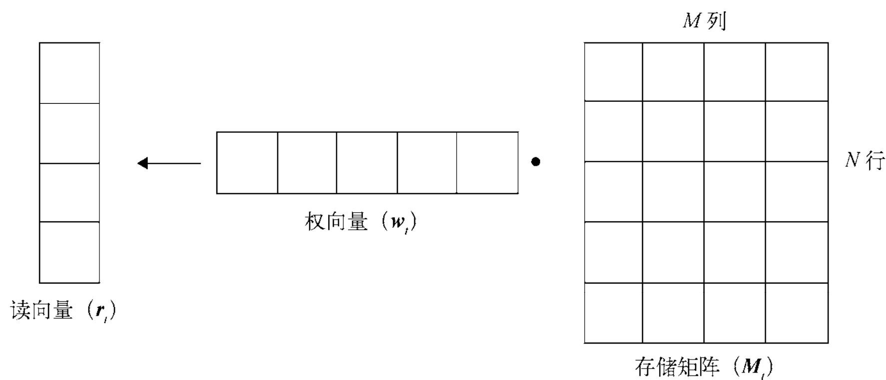



图　5-4　读操作

表达式如下：

$$\boldsymbol{r}_t <-\sum_i \boldsymbol{w}_t(i)\boldsymbol{M}_t(i)$$

可以看到，上图中有 $N$ 行和 $M$ 列的存储矩阵，以及大小为 $N$ 的权向量，其中包含所有 $N$ 个位置的权重。对它们进行线性组合，得到长度为 $M$ 的读向量。

### 2. 写操作

与读操作不同，写操作由两个子操作组成：擦除（erase）和添加（add）。这两个子操作分别擦除旧信息和向存储器添加新信息。

- **擦除操作**

我们使用擦除操作来删除存储器中不需要的信息后，会得到更新后的存储矩阵，其中原有的一些元素会被擦除。如何删除存储矩阵中特定单元格的值呢？这里引入擦除向量 $\boldsymbol{e}_t$，它和权向量 $\boldsymbol{w}_t$ 一样长，由 0 和 1 组成。

我们已有了擦除向量，但如何擦除值并得到更新后的存储矩阵呢？将 $(1-\boldsymbol{w}_t\boldsymbol{e}_t)$ 乘以上一步的存储矩阵 $\boldsymbol{M}_{t-1}$，即可得到更新后的存储矩阵 $\boldsymbol{M}_t^*$。

也就是

$$\boldsymbol{M}_t^*(i) <-(1-\boldsymbol{w}_t(i)\boldsymbol{e})\boldsymbol{M}_{t-1}(i)$$

它是如何起作用的呢？只有当权重和索引 $i$ 处的擦除元素都为 1 时，存储器中的特定元素才会被设置为 0，即被擦除；否则，它将保留自己的值。如图 5-5 所示，首先将权向量 $\boldsymbol{w}_t$ 与擦除向量 $\boldsymbol{e}_t$ 相乘。

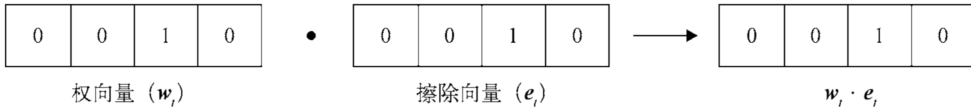



图　5-5

然后，用 1 减去它，即 $(1-\boldsymbol{w}_t(i)\boldsymbol{e}_t)$，得到一个新的向量，如图 5-6 所示。

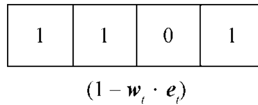



图　5-6

然后，将 $(1-\boldsymbol{w}_t\boldsymbol{e}_t)$ 乘以上一步的存储矩阵 $\boldsymbol{M}_{t-1}$，得到更新后的存储矩阵 $\boldsymbol{M}_t^*$，如图 5-7 所示。

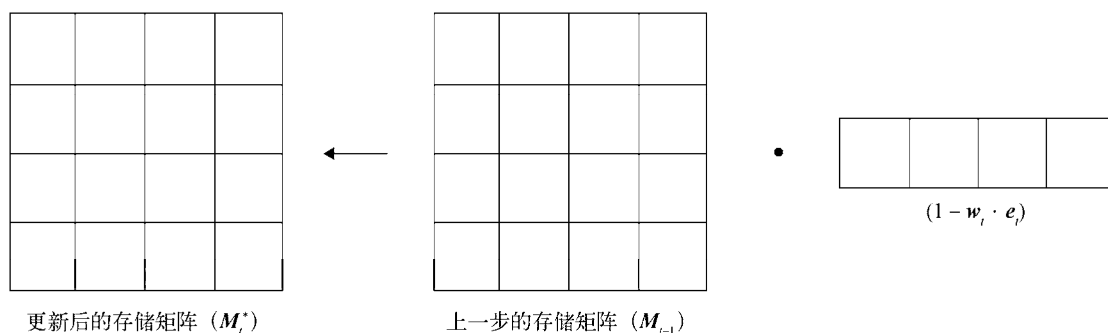



图 5-7 擦除操作

● **添加操作**

完成擦除操作后，得到更新后的存储矩阵 $M_t^*$，其中存储器中的一些元素会被擦除。现在，要向存储矩阵中添加新信息，该怎么做呢？可以引入另一个向量——加向量（add vector）$a_t$，它包含要添加到存储器中的值。如图 5-8 所示，把权向量 $w_t$ 与加向量 $a_t$ 的元素相乘，然后将其与存储矩阵相加，即 $M_t(i) <-M_t^*(i)+w_t(i)a_t$。

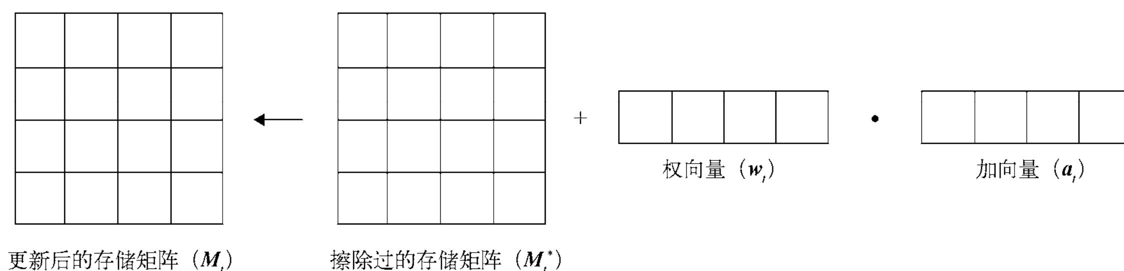



图 5-8 添加操作

## 5.1.2 寻址机制

目前为止，我们已了解了如何执行读写操作，以及如何用权向量来执行这些操作，但如何计算这个权向量呢？使用注意力机制和不同的寻址方案来计算它。我们使用两种类型的寻址机制来访问存储器中的信息：

- 基于内容的寻址（content-based addressing）
- 基于位置的寻址（location-based addressing）

### 1. 基于内容的寻址

在基于内容的寻址中，我们根据相似性从存储器中选择值。控制器返回一个键向量（key

vector）$\boldsymbol{k}_t$。我们将这个键向量 $\boldsymbol{k}_t$ 与存储矩阵 $\boldsymbol{M}_t$ 中的每一行进行比较，以学习相似性。使用余弦相似度作为相似性度量来检验相似性，公式如下：

$$\cos[\boldsymbol{k}_t, \boldsymbol{M}_t] = \frac{\boldsymbol{k}_t \cdot \boldsymbol{M}_t}{|\boldsymbol{k}_t| \cdot |\boldsymbol{M}_t|}$$

我们引入了一个新参数——键强度（key strength）$\beta$。它决定了权向量的集中程度。根据 $\beta$ 的值，可以增加或减少焦点（focus），也就是说，可以根据键强度 $\beta$ 的值调整我们对特定位置的注意力。当 $\beta$ 的值较小时，所有位置的注意力相当；当 $\beta$ 的值较大时，注意力集中在特定的位置。

因此，权向量变成如下表达式：

$$\boldsymbol{w}_t^c = \beta_t \cos[\boldsymbol{k}_t, \boldsymbol{M}_t(i)]$$

即，键向量 $\boldsymbol{k}_t$ 与存储矩阵 $\boldsymbol{M}_t$ 之间的余弦相似度，乘以键强度 $\beta$。$\boldsymbol{w}_t^c$ 中的上标 $c$ 表示它们是基于内容的权重。我们不直接使用它，而是对权重应用 softmax。因此，最终权重如下：

$$\boldsymbol{w}_t^c = \frac{\exp(\beta_t \cos[\boldsymbol{k}_t, \boldsymbol{M}_t(i)])}{\sum_j \exp(\beta_t \cos[\boldsymbol{k}_t, \boldsymbol{M}_t(j)])}$$

### 2. 基于位置的寻址

与基于内容的寻址不同，在基于位置的寻址中，我们关注的是位置而不是内容相似度。它包括 3 个步骤：

(1) 插值（interpolation）
(2) 卷积位移（convolution shift）
(3) 锐化（sharpening）

- **插值**

第一步称为插值。它用于决定是使用上一个时刻得到的权重 $\boldsymbol{w}_{t-1}$，还是使用基于内容的寻址得到的权重 $\boldsymbol{w}_t^c$。这该如何决定呢？我们使用一个新的标量参数 $g_t$，用于确定应该使用哪些权重。$g_t$ 的值是 0 或 1。

权向量的计算表达式如下：

$$\boldsymbol{w}_t^g <- g_t \boldsymbol{w}_t^c + (1 - g_t)\boldsymbol{w}_{t-1}$$

- 当 $g_t$ 为 0 时，方程变成 $\boldsymbol{w}_t^g <- \boldsymbol{w}_{t-1}$，这意味着权向量由上一个时刻得到。
- 当 $g_t$ 为 1 时，方程变成 $\boldsymbol{w}_t^g <- \boldsymbol{w}_t^c$，这意味着权向量由基于内容的寻址得到。

因此 $g_t$ 的值就像一个阀门，用来切换不同的权重。

● **卷积位移**

下一步是卷积位移。它用于移动头位置（head position），也就是说，将焦点从一个位置移动到另一个位置。每个头产生一个参数——位移权重 $\boldsymbol{s}_t$，位移权重给出允许整数位移的分布区间。例如，假设允许在 −1~1 范围内移动，那么 $\boldsymbol{s}_t$ 的长度会变为 3，并由{−1,0,1}组成。

因此，这些位移究竟意味着什么呢？假设权向量 $\boldsymbol{w}_t^g$ 中有 3 个元素，即 $\boldsymbol{w}_t^g=[\boldsymbol{w}_{i-1}^g,\boldsymbol{w}_i^g,\boldsymbol{w}_{i+1}^g]$，位移向量的中也有 3 个元素，即 $\boldsymbol{s}_t=[-1,0,1]$。

位移 −1 意味着把元素从左到右移动。位移 0 使元素保持在相同的位置，位移+1 意味着把元素从右向左移动，如图 5-9 所示。

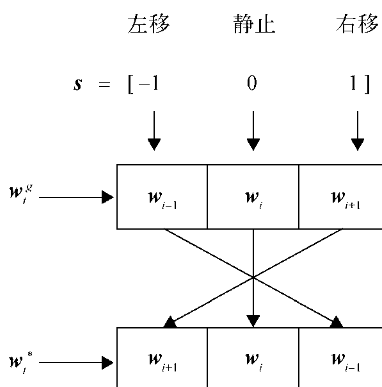



图 5-9

在图 5-10 中，位移权重 $\boldsymbol{s}_t=[1,0,0]$，意味着我们进行左移，这是因为位移值在其他位置为 0。

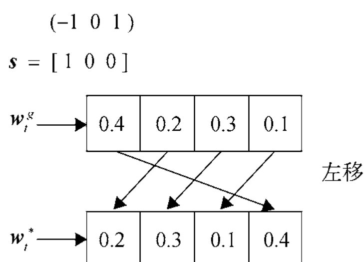



图 5-10

同样，当 $\boldsymbol{s}_t=[0,0,1]$时，我们进行右移，因为位移值在其他位置为 0，如图 5-11 所示。

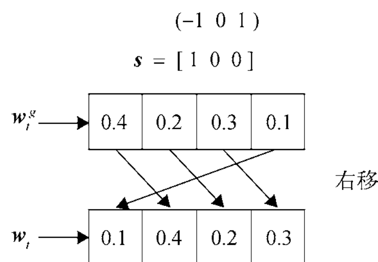



图 5-11

这样就对权重矩阵中的元素进行了卷积位移。如果存储位置为 0~$N$−1，那么可以将卷积位移表示为如下：

$$\boldsymbol{w}_t^*(i) <- \sum_{j=0}^{N-1} \boldsymbol{w}_t^g(j)\boldsymbol{s}_t(i-j)$$

- **锐化**

最后一步叫作锐化。由于卷积位移，权重 $\boldsymbol{w}_t^*$ 不会很尖锐；换句话说，由于位移，集中在一个位置的权重会分散到其他位置。为了减轻这种影响，需要进行锐化。使用一个新的参数 $\gamma_t \geqslant 1$ 来进行锐化，表达式如下：

$$\boldsymbol{w}_t(i) <- \frac{\boldsymbol{w}_t^*(i)^{\gamma_t}}{\sum_j \boldsymbol{w}_t^*(j)^{\gamma_t}}$$

## 5.2 使用 NTM 复制任务

本节我们学习如何使用 NTM 执行复制任务。复制任务的目标是观察 NTM 如何存储和回收任意长度的序列。我们将向网络提供一个随机序列以及一个表示序列结束的标记。网络必须学会输出给定的输入序列，因此，它将把输入序列存储在存储器中，然后从存储器中读取。现在，我们将逐步了解如何执行复制任务，并会在最后看到完整代码。

你也可以在 GitHub 网站搜索 Hands-On-Meta-Learning-With-Python，在带有注释的 Jupyter Notebook 中查看相应代码。

首先将了解如何实现 NTM 信元（NTM cell），我们逐步来看一下。

定义 `NTMCell` 类，用于实现完整的 NTM 信元：

```
class NTMCell():
```

定义`__init__`方法，用于初始化所有变量：

```
    def __init__(self, rnn_size, memory_size, memory_vector_dim,
read_head_num, write_head_num,
                 addressing_mode='content_and_location', shift_range=1,
reuse=False, output_dim=None):
        #初始化所有变量
        self.rnn_size = rnn_size
        self.memory_size = memory_size
        self.memory_vector_dim = memory_vector_dim
        self.read_head_num = read_head_num
        self.write_head_num = write_head_num
        self.addressing_mode = addressing_mode
        self.reuse = reuse
        self.step = 0
        self.output_dim = output_dim
        self.shift_range = shift_range
        #初始化控制器作为基本 RNN 单元
        self.controller = tf.nn.rnn_cell.BasicRNNCell(self.rnn_size)
```

定义`__call__`方法，用于实现 NTM 的操作：

```
def __call__(self, x, prev_state):
```

通过将输入 `x` 与之前读取的向量列表结合，获得控制器的输入：

```
prev_read_vector_list = prev_state['read_vector_list']
prev_controller_state = prev_state['controller_state']

controller_input = tf.concat([x] + prev_read_vector_list, axis=1)
```



通过将 `controller_input` 和 `prev_controller_state` 输入，我们构建控制器，即 RNN 单元：

```
with tf.variable_scope('controller', reuse=self.reuse):
    controller_output, controller_state = self.controller(controller_input,
prev_controller_state)
```

初始化读头与写头：

```
num_parameters_per_head = self.memory_vector_dim + 1 + 1 +
(self.shift_range * 2 + 1) + 1
num_heads = self.read_head_num + self.write_head_num
total_parameter_num = num_parameters_per_head * num_heads +
self.memory_vector_dim * 2 * self.write_head_num
```

初始化权重矩阵以及偏置（bias），并使用前馈操作计算参数：

```
with tf.variable_scope("o2p", reuse=(self.step > 0) or self.reuse):
    o2p_w = tf.get_variable('o2p_w', [controller_output.get_shape()[1],
total_parameter_num],
initializer=tf.random_normal_initializer(mean=0.0, stddev=0.5))
    o2p_b = tf.get_variable('o2p_b', [total_parameter_num],
initializer=tf.random_normal_initializer(mean=0.0, stddev=0.5))
    parameters = tf.nn.xw_plus_b(controller_output, o2p_w, o2p_b)
```

```
head_parameter_list = tf.split(parameters[:, :num_parameters_per_head *
num_heads], num_heads, axis=1)
erase_add_list = tf.split(parameters[:, num_parameters_per_head *
num_heads:], 2 * self.write_head_num, axis=1)
```

得到上一步的权向量与存储器：

```
#上一步的权向量
prev_w_list = prev_state['w_list']

#上一步的存储器
prev_M = prev_state['M']

w_list = []
p_list = []
```

现在初始化一些用于寻址的重要参数：

```
for i, head_parameter in enumerate(head_parameter_list):

    #键向量
    k = tf.tanh(head_parameter[:, 0:self.memory_vector_dim])

    #键强度 (beta)
    beta = tf.sigmoid(head_parameter[:, self.memory_vector_dim]) * 10

    #插值门
    g = tf.sigmoid(head_parameter[:, self.memory_vector_dim + 1])

    #移位矩阵
    s = tf.nn.softmax(
        head_parameter[:, self.memory_vector_dim + 2:self.memory_vector_dim
+ 2 + (self.shift_range * 2 + 1)]
    )

    #锐化因子
    gamma = tf.log(tf.exp(head_parameter[:, -1]) + 1) + 1

    with tf.variable_scope('addressing_head_%d' % i):
        w = self.addressing(k, beta, g, s, gamma, prev_M, prev_w_list[i])

    w_list.append(w)
    p_list.append({'k': k, 'beta': beta, 'g': g, 's': s, 'gamma': gamma})
```

## 读操作

选择读头：

```
read_w_list = w_list[:self.read_head_num]
```

读操作是权重和存储器的线性组合：

```
read_vector_list = []
for i in range(self.read_head_num):
```

```
    #权重和存储器的线性组合
    read_vector = tf.reduce_sum(tf.expand_dims(read_w_list[i], dim=2) *
prev_M, axis=1)
    read_vector_list.append(read_vector)
```

### 写操作

与读操作不同，写操作包括擦除与添加两步。

下列代码选取了一个写头：

```
write_w_list = w_list[self.read_head_num:]

#更新存储器
M = prev_M
```

执行擦除与添加操作：

```
for i in range(self.write_head_num):

    #擦除向量将与权向量相乘，以表示要擦除或保持不变的位置
    w = tf.expand_dims(write_w_list[i], axis=2)
    erase_vector = tf.expand_dims(tf.sigmoid(erase_add_list[i * 2]),
axis=1)

    #接下来执行添加操作
    add_vector = tf.expand_dims(tf.tanh(erase_add_list[i * 2 + 1]), axis=1)
    M = M * (tf.ones(M.get_shape()) - tf.matmul(w, erase_vector)) +
tf.matmul(w, add_vector)
```



获取控制器输出：

```
if not self.output_dim:
    output_dim = x.get_shape()[1]
else:
    output_dim = self.output_dim

with tf.variable_scope("o2o", reuse=(self.step > 0) or self.reuse):
    o2o_w = tf.get_variable('o2o_w', [controller_output.get_shape()[1],
output_dim],
initializer=tf.random_normal_initializer(mean=0.0, stddev=0.5))
    o2o_b = tf.get_variable('o2o_b', [output_dim],
initializer=tf.random_normal_initializer(mean=0.0, stddev=0.5))
    NTM_output = tf.nn.xw_plus_b(controller_output, o2o_w, o2o_b)

state = {
    'controller_state': controller_state,
    'read_vector_list': read_vector_list,
    'w_list': w_list,
    'p_list': p_list,
    'M': M
}

self.step += 1
```

### 寻址机制

我们使用两种寻址机制：基于内容的寻址与基于位置的寻址。

### 基于内容的寻址

计算键向量与存储矩阵之间的余弦相似度：

```
k = tf.expand_dims(k, axis=2)
inner_product = tf.matmul(prev_M, k)

k_norm = tf.sqrt(tf.reduce_sum(tf.square(k), axis=1, keepdims=True))
M_norm = tf.sqrt(tf.reduce_sum(tf.square(prev_M), axis=2, keepdims=True))
norm_product = M_norm * k_norm

K = tf.squeeze(inner_product / (norm_product + 1e-8))
```

现在，根据相似度和键强度（`beta`）生成归一化权向量。`beta` 用于调整头部对焦精度：

```
K_amplified = tf.exp(tf.expand_dims(beta, axis=1) * K)
w_c = K_amplified / tf.reduce_sum(K_amplified, axis=1, keepdims=True) # eq
(5)
```

### 基于位置的寻址

基于位置的寻址包括 3 个步骤：

(1) 插值
(2) 卷积位移
(3) 锐化

### 插值

这用于决定是应该使用前一个时间步骤获得的权重 `prev_w`，还是使用基于内容的寻址获得的权重 `w_c`。可该如何决定呢？我们使用一个新的标量参数 `g`，来决定应该使用哪些权重：

```
g = tf.expand_dims(g, axis=1)
w_g = g * w_c + (1 - g) * prev_w
```

### 卷积位移

插值后进行卷积位移，使控制器能够集中于其他行：

```
s = tf.concat([s[:, :self.shift_range + 1],
               tf.zeros([s.get_shape()[0], self.memory_size -
(self.shift_range * 2 + 1)]),
               s[:, -self.shift_range:]], axis=1)

t = tf.concat([tf.reverse(s, axis=[1]), tf.reverse(s, axis=[1])], axis=1)
```

```
s_matrix = tf.stack(
    [t[:, self.memory_size - i - 1:self.memory_size * 2 - i - 1] for i in
range(self.memory_size)],
    axis=1
)

w_ = tf.reduce_sum(tf.expand_dims(w_g, axis=1) * s_matrix, axis=2) # eq (8)
```

**锐化**

卷积位移后进行锐化操作，以避免位移后的权向量变模糊：

```
w_sharpen = tf.pow(w_, tf.expand_dims(gamma, axis=1))
w = w_sharpen / tf.reduce_sum(w_sharpen, axis=1, keepdims=True)
```

然后定义一个名为 `zero_state` 的函数，用于初始化控制器、读向量、权重和存储器的所有状态：

```
    def zero_state(self, batch_size, dtype):
        def expand(x, dim, N):
            return tf.concat([tf.expand_dims(x, dim) for _ in range(N)],
axis=dim)
        with tf.variable_scope('init', reuse=self.reuse):
            state = {
                'controller_state':
expand(tf.tanh(tf.get_variable('init_state', self.rnn_size,
initializer=tf.random_normal_initializer(mean=0.0, stddev=0.5))),
                                  dim=0, N=batch_size),
                'read_vector_list':
[expand(tf.nn.softmax(tf.get_variable('init_r_%d' % i,
[self.memory_vector_dim],
initializer=tf.random_normal_initializer(mean=0.0, stddev=0.5))),
                                  dim=0, N=batch_size)
                           for i in range(self.read_head_num)],
                'w_list': [expand(tf.nn.softmax(tf.get_variable('init_w_%d'
% i, [self.memory_size],
initializer=tf.random_normal_initializer(mean=0.0, stddev=0.5))),
                                  dim=0, N=batch_size) if
self.addressing_mode == 'content_and_loaction'
                           else tf.zeros([batch_size, self.memory_size])
                           for i in range(self.read_head_num +
self.write_head_num)],
                'M': expand(tf.tanh(tf.get_variable('init_M',
[self.memory_size, self.memory_vector_dim],
initializer=tf.random_normal_initializer(mean=0.0, stddev=0.5))),
                                  dim=0, N=batch_size)
            }
            return state
```

接下来定义一个名为 `generate_random_string` 的函数，它生成一个长度为 `seq_length` 的随机序列，我们将这些序列作为 NTM 的输入，用于复制任务：

```
def generate_random_strings(batch_size, seq_length, vector_dim):
    return np.random.randint(0, 2, size=[batch_size, seq_length,
vector_dim]).astype(np.float32)
```

现在，创建 NTMCopyModel 来执行整个复制任务：

```
class NTMCopyModel():
    def __init__(self, args, seq_length, reuse=False):
        #输入序列
        self.x = tf.placeholder(name='x', dtype=tf.float32,
shape=[args.batch_size, seq_length, args.vector_dim])
        #输出序列
        self.y = self.x
        #序列末尾
        eof = np.zeros([args.batch_size, args.vector_dim + 1])
        eof[:, args.vector_dim] = np.ones([args.batch_size])
        eof = tf.constant(eof, dtype=tf.float32)
        zero = tf.constant(np.zeros([args.batch_size, args.vector_dim +
1]), dtype=tf.float32)
        if args.model == 'LSTM':
            def rnn_cell(rnn_size):
                return tf.nn.rnn_cell.BasicLSTMCell(rnn_size, reuse=reuse)
            cell = tf.nn.rnn_cell.MultiRNNCell([rnn_cell(args.rnn_size) for
_ in range(args.rnn_num_layers)])
        elif args.model == 'NTM':
            cell = NTMCell(args.rnn_size, args.memory_size,
args.memory_vector_dim, 1, 1,
                                    addressing_mode='content_and_location',
                                    reuse=reuse,
                                    output_dim=args.vector_dim)
        #初始化所有状态
        state = cell.zero_state(args.batch_size, tf.float32)
        self.state_list = [state]
        for t in range(seq_length):
            output, state = cell(tf.concat([self.x[:, t, :],
np.zeros([args.batch_size, 1])], axis=1), state)
            self.state_list.append(state)
        #获取输出与状态
        output, state = cell(eof, state)
        self.state_list.append(state)

        self.o = []
        for t in range(seq_length):
            output, state = cell(zero, state)
            self.o.append(output[:, 0:args.vector_dim])
            self.state_list.append(state)
        self.o = tf.sigmoid(tf.transpose(self.o, perm=[1, 0, 2]))

        eps = 1e-8
        #将损失计算为交叉熵损失
        self.copy_loss = -tf.reduce_mean(self.y * tf.log(self.o + eps) + (1
- self.y) * tf.log(1 - self.o + eps))
        #使用 RMS prop 优化器进行优化
        with tf.variable_scope('optimizer', reuse=reuse):
```

```
            self.optimizer =
tf.train.RMSPropOptimizer(learning_rate=args.learning_rate, momentum=0.9,
decay=0.95)
            gvs = self.optimizer.compute_gradients(self.copy_loss)
            capped_gvs = [(tf.clip_by_value(grad, -10., 10.), var) for
grad, var in gvs]
            self.train_op = self.optimizer.apply_gradients(capped_gvs)
        self.copy_loss_summary = tf.summary.scalar('copy_loss_%d' %
seq_length, self.copy_loss)
```

我们使用以下命令重置 TensorFlow 图：

```
tf.reset_default_graph()
```

之后，定义所有参数：

```
parser = argparse.ArgumentParser()
parser.add_argument('--mode', default="train")
parser.add_argument('--restore_training', default=False)
parser.add_argument('--test_seq_length', type=int, default=5)
parser.add_argument('--model', default="NTM")
parser.add_argument('--rnn_size', default=16)
parser.add_argument('--rnn_num_layers', default=3)
parser.add_argument('--max_seq_length', default=5)
parser.add_argument('--memory_size', default=16)
parser.add_argument('--memory_vector_dim', default=5)
parser.add_argument('--batch_size', default=5)
parser.add_argument('--vector_dim', default=8)
parser.add_argument('--shift_range', default=1)
parser.add_argument('--num_epoches', default=100)
parser.add_argument('--learning_rate', default=1e-4)
parser.add_argument('--save_dir', default= os.getcwd())
parser.add_argument('--tensorboard_dir', default=os.getcwd())
args = parser.parse_args(args = [])
```

最后，定义 `training` 函数：

```
def train(args):
    model_list = [NTMCopyModel(args, 1)]
    for seq_length in range(2, args.max_seq_length + 1):
        model_list.append(NTMCopyModel(args, seq_length, reuse=True))

    with tf.Session() as sess:
        if args.restore_training:
            saver = tf.train.Saver()
            ckpt = tf.train.get_checkpoint_state(args.save_dir + '/' +
args.model)
            saver.restore(sess, ckpt.model_checkpoint_path)
        else:
            saver = tf.train.Saver(tf.global_variables())
            tf.global_variables_initializer().run()
        #初始化摘要编写器，以便在tensorboard 中可视化
        train_writer = tf.summary.FileWriter(args.tensorboard_dir,
sess.graph)
```

5

```
        plt.ion()
        plt.show()
        for b in range(args.num_epoches):
            #初始化序列长度
            seq_length = np.random.randint(1, args.max_seq_length + 1)
            model = model_list[seq_length - 1]
            #生成随机输入序列作为输入
            x = generate_random_strings(args.batch_size, seq_length,
args.vector_dim)
            #将随机输入序列输入模型
            feed_dict = {model.x: x}
            if b % 100 == 0:
                p = 0
                print("First training batch sample",x[p, :, :])
                #计算模型输出
                print("Model output",sess.run(model.o,
feed_dict=feed_dict)[p, :, :])
                state_list = sess.run(model.state_list,
feed_dict=feed_dict)
                if args.model == 'NTM':
                    w_plot = []
                    M_plot = np.concatenate([state['M'][p, :, :] for state
in state_list])
                    for state in state_list:
                        w_plot.append(np.concatenate([state['w_list'][0][p,
:], state['w_list'][1][p, :]]))
                    #绘制权重矩阵以观察注意力
                    plt.imshow(w_plot, interpolation='nearest',
cmap='gray')
                    plt.draw()
                    plt.pause(0.001)
                #计算损失
                copy_loss = sess.run(model.copy_loss, feed_dict=feed_dict)
                #编写摘要
                merged_summary = sess.run(model.copy_loss_summary,
feed_dict=feed_dict)
                train_writer.add_summary(merged_summary, b)
                print('batches %d, loss %g' % (b, copy_loss))
            else:
                sess.run(model.train_op, feed_dict=feed_dict)
            #保存模型
            if b % 5000 == 0 and b > 0:
                saver.save(sess, args.save_dir + '/' + args.model +
'/model.tfmodel', global_step=b)
```

我们使用下列命令，开始训练 NTM：

```
train(args)
```

输出如图 5-12，可以看到注意力集中于权重矩阵。

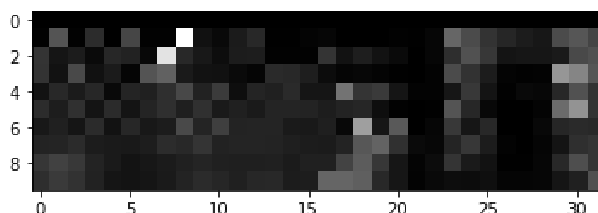



图 5-12

## 5.3 MANN

本节我们会学习一个有趣的 NTM 变体，叫作记忆增强神经网络（MANN）。它广泛用于单样本学习任务，旨在让 NTM 在单样本学习任务中表现得更好。我们知道 NTM 既可以使用基于内容的寻址，又可以使用基于位置的寻址，但是在 MANN 中，只使用基于内容的寻址。

MANN 使用了一种新的寻址方案，称为最近最少使用访问（least recently used access）。顾名思义，它将写入最近最少使用的存储器位置。等等，什么？我们刚刚了解到 MANN 并不是基于位置的，那么为什么要写最近最少使用的位置呢？这是因为最近最少使用的存储器位置由读操作决定，而读操作由基于内容的寻址执行。因此，我们基本上是执行基于内容的寻址来读写最近最少使用的位置。

### 读操作和写操作

本节我们学习如何在 MANN 中执行读操作和写操作，以及它们与 NTM 的区别。

#### 1. 读操作

与 NTM 不同，在 MANN 中，我们使用两个不同的权向量来执行读写操作。MANN 中的读操作与 NTM 相同。因为在 MANN 中使用基于内容的相似性执行读操作，所以我们将控制器发出的键向量 $\boldsymbol{k}_t$ 与存储矩阵 $\boldsymbol{M}_t$ 中的每一行进行比较，以了解相似性。我们使用余弦相似度作为相似性度量来检验相似性，表达式如下：

$$\cos[\boldsymbol{k}_t, \boldsymbol{M}_t] = \frac{\boldsymbol{k}_t \cdot \boldsymbol{M}_t}{|\boldsymbol{k}_t| \cdot |\boldsymbol{M}_t|}$$

因此，权向量变成如下的表达式：

$$\boldsymbol{w}_t^r = \cos[\boldsymbol{k}_t, \boldsymbol{M}_t(i)]$$

但与 NTM 不同的是，在这里不使用键强度 $\beta$ 。$\boldsymbol{w}_t^r$ 中的上标 $r$ 表示它是读向量。最终的权向

量是基于权重的 softmax：

$$w_t^r = \frac{\exp(\cos[\boldsymbol{k}_t, \boldsymbol{M}_t(i)])}{\sum_j \exp(\cos[\boldsymbol{k}_t, \boldsymbol{M}_t(j)])}$$

读向量是权重 $\boldsymbol{w}_t^r$ 与存储矩阵 $\boldsymbol{M}_t$ 的线性组合：

$$\boldsymbol{r}_t <- \sum_i^R \boldsymbol{w}_t^r(i)\boldsymbol{M}_t(i)$$

让我们看看如何使用 TensorFlow 构建它。

首先，使用基于内容的相似性计算读向量：

```
    def read_head_addressing(k, prev_M):
        k = tf.expand_dims(k, axis=2)
        inner_product = tf.matmul(prev_M, k)
        k_norm = tf.sqrt(tf.reduce_sum(tf.square(k), axis=1,
keep_dims=True))
        M_norm = tf.sqrt(tf.reduce_sum(tf.square(prev_M), axis=2,
keep_dims=True))
        norm_product = M_norm * k_norm
        K = tf.squeeze(inner_product / (norm_product + 1e-8))
        K_exp = tf.exp(K)
        w = K_exp / tf.reduce_sum(K_exp, axis=1, keep_dims=True)
        return w
```

然后，得到读权向量（read weight vector）：

```
w_r = read_head_addressing(k, prev_M)
```

执行读操作，即读权向量与存储器的线性组合：

```
  read_vector_list = []
        with tf.variable_scope('reading'):
            for i in range(self.head_num):
                read_vector = tf.reduce_sum(tf.expand_dims(w_r_list[i],
dim=2) * M, axis=1)
                read_vector_list.append(read_vector)
```

### 2. 写操作

在执行写操作之前，我们希望找到最近最少使用的存储器位置，因为这是我们必须写入的位置。如何找到最近最少使用的存储器位置呢？为了找到它，我们计算一个新的向量——用途权向量 $\boldsymbol{w}_t^u$，它将在每次读写之后更新，且只是读权向量和写权向量的和，即 $\boldsymbol{w}_t^u <- \boldsymbol{w}_t^r + \boldsymbol{w}_t^w$。

除了添加读向量和权向量外，我们还通过添加衰减的上一阶段的用途权向量 $\boldsymbol{w}_{t-1}^u$ 来更新用途权向量。我们使用衰减参数 $\gamma$，该参数用于确定上一阶段的用途权重是如何衰减的。因此，最终的用途权向量是上一阶段用途权向量的衰减、读权向量、写权向量的和：

$$w_t^u < -\gamma w_{t-1}^u + w_t^r + w_t^w$$

现在已经计算了用途权向量，如何计算最近最少使用的位置呢？为此，我们引入了另一个权向量——最少用途权向量（least used weight vector）$w_t^{lu}$。

由用途权向量 $w_t^u$ 计算最少使用权向量 $w_t^{lu}$ 非常简单。只需将用途权向量中最小值的索引设置为 1，其他值设置为 0 即可，因为用途权向量中最小值是最近最少使用的（见图 5-13）。

| 0.3 | 0.3 | 0.1 | 0.3 |
|---|---|---|---|

用途权向量（$w_t^u$）

| 0 | 0 | 1 | 0 |
|---|---|---|---|

最近最少使用权向量（$w_t^{lu}$）

图　5-13

好了，接下来是什么呢？我们已计算了最近最少使用权向量。现在，如何计算写权向量 $w_t^w$ 呢？我们使用 sigmoid 门计算写权向量，sigmoid 门用于计算前一个读权向量 $w_{t-1}^r$ 和前一个最近最少使用权向量 $w_{t-1}^{lu}$ 的凸组合：

$$w_t^w < -\sigma(\alpha) w_{t-1}^r + (1-\sigma(\alpha)) w_{t-1}^{lu}$$

计算写权向量后，最后更新存储矩阵：

$$M_t(i) < -M_{t-1}(i) + w_t^w(i) k_t$$

下面是如何在 TensorFlow 中构建这个模型。

首先，计算用途权向量：

```
w_u = self.gamma * prev_w_u + tf.add_n(w_r_list) + tf.add_n(w_w_list)
```

然后，计算最近最少使用权向量：

```
    def least_used(w_u):
        _, indices = tf.nn.top_k(w_u, k=self.memory_size)
        w_lu = tf.reduce_sum(tf.one_hot(indices[:, -self.head_num:],
depth=self.memory_size), axis=1)
        return indices, w_lu
```

存储上一步的索引和最近最少使用权向量：

```
prev_indices, prev_w_lu = least_used(prev_w_u)
```

计算写权向量：

```
def write_head_addressing(sig_alpha, prev_w_r, prev_w_lu):
    return sig_alpha * prev_w_r + (1. - sig_alpha) * prev_w_lu
```

之后，更新存储矩阵：

```
M_ = prev_M * tf.expand_dims(1. - tf.one_hot(prev_indices[:, -1],
self.memory_size), dim=2)
```

执行写操作：

```
M = M_
with tf.variable_scope('writing'):
    for i in range(self.head_num):
        w = tf.expand_dims(w_w_list[i], axis=2)
        k = tf.expand_dims(k_list[i], axis=1)
        M = M + tf.matmul(w, k)
```

## 5.4　小结

在本章，我们首先了解了 NTM 如何向存储器中存储和从中检索信息，以及它如何使用不同的寻址机制（例如基于位置和基于内容的寻址）来读写信息；然后学习了如何使用 TensorFlow 来实现 NTM，以执行复制任务；之后探讨了 MANN 以及 MANN 与 NTM 的不同之处；最后研究了 MANN 如何使用最近最少使用的访问方法来克服 NTM 的缺点。

下一章将介绍**模型无关元学习**及其在监督和强化学习场景中的应用。

## 5.5　思考题

(1) 什么是 NTM？
(2) NTM 中的控制器是什么？
(3) 为什么使用读头与写头？
(4) 什么是存储器？
(5) NTM 中有哪些不同的寻址方式？
(6) 什么是插值门？
(7) 如何由用途权向量计算出最近最少使用权向量？

## 5.6　延伸阅读

- NTM 论文：Alex Graves、Greg Wayne 和 Ivo Danihelka 的文章 *Neural Turing Machines*。
- 使用记忆增强神经网络进行单样本学习：参见 Adam Santoro、Sergey Bartunov、Matthew Botvinick 等人的文章 *One-shot Learning with Memory-Augmented Neural Networks*。

第 6 章

# MAML 及其变种

在前一章，我们学习了 NTM 及其如何向存储器中存储和从中检索信息，还了解了 NTM 的变体——MANN，它广泛用于单样本学习。在本章，我们将学习一个有趣且非常常用的元学习算法——**模型无关元学习**（model agnostic meta learning，MAML）；了解什么是 MAML，以及它在监督和强化学习场景中是如何使用的；分析如何从头开始构建 MAML，并学习**对抗元学习**（adversarial meta learning，ADML）；阐释如何使用 ADML 来找到一个稳健的模型参数，以及如何为分类任务实现 ADML；掌握**元学习的上下文适应**（context adaptation for meta learning，CAML）。

本章内容包括：

- MAML；
- MAML 算法；
- 监督和强化学习场景中的 MAML；
- 从头构建 MAML；
- ADML；
- 从头构建 ADML；
- CAML。



## 6.1 MAML

MAML 是近年来引入的、应用最广泛的元学习算法之一，在元学习研究方面取得了重大突破。学会学习是元学习的重点，在元学习中，我们只从少量数据点中学习各种相关任务，与此同时元学习器可以生成快速的学习器，即使在新的训练样本较少的相关任务中也有较好的泛化效果。

MAML 的基本思想是寻找一个更好的初始参数，这样，在初始参数良好的情况下，模型可以以较少的梯度步骤（gradient step）快速学习新任务。

这是什么意思呢？假设我们正在使用神经网络执行分类任务。该如何训练网络呢？从初始化随机权重开始，通过最小化损失来训练网络。怎样才能最小化损失呢？使用梯度下降。但如何使用梯度下降来最小化损失呢？使用梯度下降来寻找最优权重，使损失最小。通过多个梯度步骤来寻找最优权重，从而达到收敛。

在 MAML 中，我们试图通过学习相似任务的分布来找到这些最优权重。因此，对于一个新任务，不需要从随机初始化的权重开始；相反，可以从最优权重开始，它将采取更少的梯度步骤来达到收敛，并且不需要较多的数据点来进行训练。

让我们换种简单的方式来理解 MAML。假设有 3 个相关的任务：$T_1$、$T_2$ 和 $T_3$。首先，随机初始化模型参数 $\theta$。我们在任务 $T_1$ 上训练网络，然后，尝试通过梯度下降使损失最小化。通过寻找最优参数 $\theta_1'$，使损失最小化。同样，我们也会通过随机初始化模型参数 $\theta$，来训练 $T_2$ 和 $T_3$，通过梯度下降找到正确的参数组合来最小化损失。假设 $\theta_2'$ 和 $\theta_3'$ 分别为任务 $T_2$ 和 $T_3$ 的最优参数。

如图 6-1 所示，首先随机初始化模型参数 $\theta$，然后通过寻找最优参数 $\theta_1'$、$\theta_2'$ 和 $\theta_3'$，为 $T_1$、$T_2$ 和 $T_3$ 分别最小化损失。

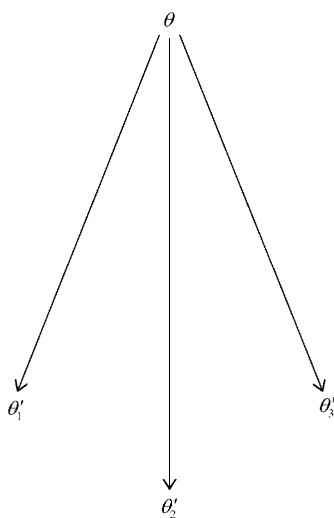



图　6-1

然而，除了将 $\theta$ 初始化在随机位置（即用随机值初始化 $\theta$）以外，如果能够在某个位置初始化 $\theta$，使得它对 3 个任务都通用，那么梯度步骤与训练时间就会减少。这正是 MAML 试图做到的。MAML 试图找到这个在许多相关的任务中共同的最优参数 $\theta$，从而我们可以相对迅速地使用少量数据点训练新任务且无须很多梯度步骤。

如图 6-2 所示，将$\theta$移动到对所有不同的最优$\theta'$值都通用的一个位置。

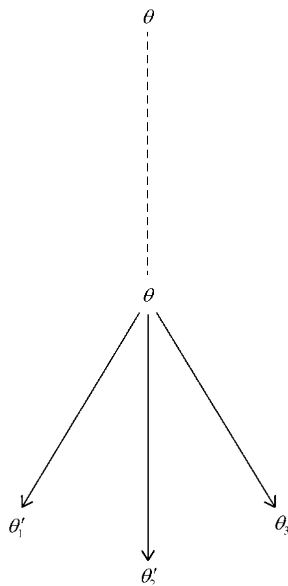



图 6-2

因此，对于新的相关任务$T_4$，无须随机初始化参数$\theta$。可以从最优$\theta$值出发，它会使用较少的梯度步骤达到收敛。

因此，在 MAML 中，我们试图找到的这个最优$\theta$值，它对相关任务是通用的，这将帮助我们从较少的数据点中学习，并减少训练时间。MAML 与模型无关，这意味着我们可以将 MAML 应用于任何可以通过梯度下降训练的模型，但它究竟是如何运作的？如何将模型参数移动到最优位置？下一节将详细探讨。

## 6.1.1 MAML 算法

现在我们已基本了解了 MAML，接下来会详细地探讨它。假设有一个由$\theta$影响的模型$f_\theta()$以及任务的分布$p(T)$。首先，用一些随机值初始化参数$\theta$；接下来，从任务的分布中抽取一批任务$T_i$（即$T_i \sim p(T)$）。假设抽取了 5 个任务$T=\{T_1,T_2,T_3,T_4,T_5\}$，然后，对于每个任务$T_i$，抽取$k$个数据点并训练模型。通过计算损失$L_{T_i}(f_\theta)$，利用梯度下降最小化损失，找到使损失最小化的最优参数集：

$$\theta_i' = \theta - \alpha \nabla_\theta L_{T_i}(f_\theta)$$

参数解释如下：

- $\theta_i'$ 是任务 $T_i$ 的最优参数；
- $\theta$ 是初始参数；
- $\alpha$ 是超参数；
- $\nabla_\theta L_{T_i}(f_\theta)$ 是任务 $T_i$ 的梯度。

因此，在之前的梯度更新后，采样的 5 个任务都有最优参数：

$$\theta' = \{\theta_1', \theta_2', \theta_3', \theta_4', \theta_5'\}$$

在抽取下一批任务之前，执行元更新或元优化。也就是说，在前面的步骤中，通过对每个任务 $T_i$ 的训练找到了最优参数 $\theta_i'$。现在计算相对于这些最优参数 $\theta_i'$ 的梯度，并通过在新一批任务 $T_i$ 上训练，更新随机初始化参数 $\theta$。这使得随机初始化参数 $\theta$ 移动到最佳位置，在该位置上训练下一批任务时，无须花费很多梯度步骤。此步骤称为元步骤、元更新、元优化或元训练，表达式如下：

$$\theta = \theta - \beta \nabla_\theta \sum_{T_i \sim p(T)} L_{T_i}(f_{\theta_i'})$$

参数解释如下：

- $\theta$ 是初始参数；
- $\beta$ 是超参数；
- $\nabla_\theta \sum_{T_i \sim p(T)} L_{T_i}(f_{\theta_i'})$ 是每个新任务 $T_i$ 相对于参数 $\theta_i'$ 的梯度。

如果仔细观察之前的元更新方程，可以发现，我们仅仅通过对每个新任务 $T_i$（其最优参数为 $\theta_i'$）的梯度取平均数来更新模型参数 $\theta$。

MAML 的整体算法如图 6-3 所示。算法包括两个循环：内循环和外循环。在内循环中，找到每个任务 $T_i$ 的最优参数 $\theta_i'$。在外循环中，通过计算每个新任务 $T_i$ 中相对于最优参数 $\theta_i'$ 的梯度，更新随机初始化的模型参数 $\theta$。

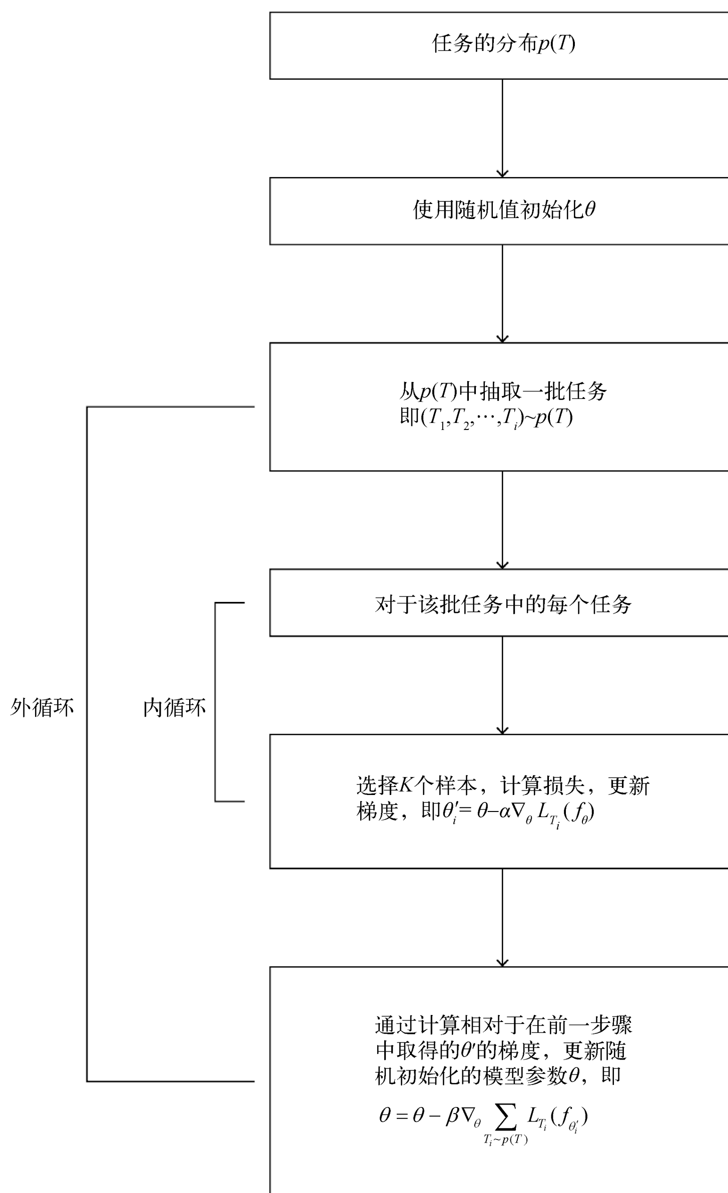



图 6-3



我们应该永远记住，不应该使用同一组任务$T_i$的最优参数$\theta_i'$，来在外循环中更新模型参数$\theta$。

因此，简而言之，在 MAML 中，我们抽取一批任务，对于这批任务中的每个任务$T_i$，使用

梯度下降最小化损失，并得到最优参数 $\theta'_i$ 。然后，在抽样另一个批任务之前，通过计算每个新任务 $T_i$ 中相对于最优参数 $\theta'_i$ 的梯度，更新随机初始化模型参数 $\theta$ 。

## 6.1.2　监督学习中的 MAML

如你所知，MAML 擅于找到最优初始参数。下面来看看如何在监督学习场景中使用 MAML。让我们先快速定义损失函数。根据执行的任务，损失函数可以是任何函数。

如果要进行回归，那么可以使用均方误差作为损失函数：

$$L_{T_i}(f_\theta)=\sum_{x_j,y_j\sim T_i}\|f_\theta(x_i)-y_i\|_2^2$$

如果是分类任务，则可以使用交叉熵损失作为损失函数：

$$L_{T_i}(f_\theta)=\sum_{x_j,y_j\sim T_i}y_j\log f_\theta(x_j)+(1-y_j)\log(1-f_\theta(x_j))$$

下面逐步分析 MAML 在监督学习中的运用。

(1) 假设有一个由 $\theta$ 影响的模型 $f_\theta$ 以及任务的分布 $p(T)$ 。首先，随机初始化参数 $\theta$ 。
(2) 从任务的分布中抽取一批任务 $T_i$（即 $T_i\sim p(T)$ ）。假设抽取了 3 个任务 $T=\{T_1,T_2,T_3\}$ 。
(3) **内循环**：对于任务 $T$ 中的每个任务 $T_i$ ，抽取 $k$ 个数据点并准备训练集与测试集：

$$D_i^{\text{train}}=\{(x_1,y_1),(x_2,y_2),\cdots,(x_k,y_k)\}$$
$$D_i^{\text{test}}=\{(x_1,y_1),(x_2,y_2),\cdots,(x_k,y_k)\}$$

等等！训练集和测试集是什么？我们在内循环中使用训练集寻找最优参数 $\theta'_i$ ，在外循环中使用测试集寻找最优参数 $\theta$ 。**测试集并不意味着我们在检查模型的性能，它基本上就像外循环中的训练集。也可以将测试集称为元训练集。**

现在我们在 $D_i^{\text{train}}$ 上应用监督学习算法，利用梯度下降计算损失并使损失最小化，得到最优参数 $\theta'_i$ ，使得 $\theta'_i=\theta-\alpha\nabla_\theta L_{T_i}(f_\theta)$ 。因此，对于每个任务，抽取 $k$ 个数据点，最小化训练集 $D_i^{\text{train}}$ 上的损失，得到最优参数 $\theta'_i$ 。当抽取 3 个任务时，我们会有 3 个最优参数 $\{\theta'_1,\theta'_2,\theta'_3\}$ 。

(4) **外循环**：我们在测试集（元训练集）中执行元优化。这里试图使测试集 $D_i^{\text{test}}$ 中的损失最小化。通过计算相对于上一步最优参数 $\theta'_i$ 的梯度以减少损失，并使用测试集（元训练集）更新随机初始化参数 $\theta$ ：

$$\theta=\theta-\beta\nabla_\theta\sum_{T_i\sim p(T)}L_{T_i}(f_{\theta'_i})$$

(5) 我们对步骤(2)到步骤(5)进行 $n$ 次迭代。图 6-4 概括了监督学习中的 MAML。

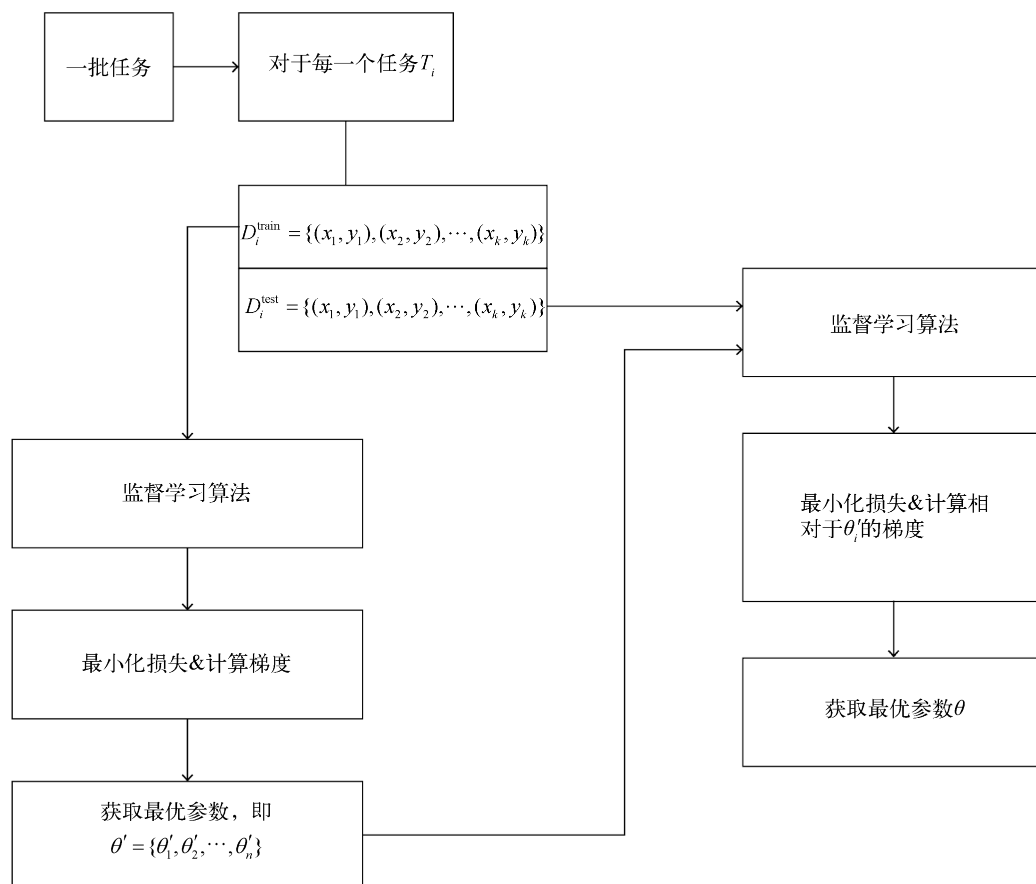



图 6-4

### 从头构建 MAML

在上一节，我们了解了 MAML 的工作原理及其是如何获得更好、更稳健的模型参数$\theta$，以在任务间泛化。现在，为了更好地理解 MAML，我们将从头开始编写代码，并考虑一个简单的二分类任务。我们随机生成输入数据并使用简单的单层神经网络训练它，试图找到最优参数$\theta$。下面会一步一步地讲解如何做到这一点。

你也可以在 GitHub 网站搜索 Hands-On-Meta-Learning-With-Python，在带有注释的 Jupyter Notebook 中查看相应代码。

首先，导入 `numpy` 库：

```
import numpy as np
```

- **生成数据点**

现在定义一个名为 `sample_points` 的函数来生成输入对(x, y)。它以参数 k 为输入，表示要抽取的(x, y)对的个数：

```
def sample_points(k):
    x = np.random.rand(k,50)
    y = np.random.choice([0, 1], size=k, p=[.5, .5]).reshape([-1,1])
    return x,y
```

上述函数输出如下：

```
x, y = sample_points(10)
print x[0]
print y[0]

[0.537339   0.113621   0.62983308  0.3016117   0.91174146  0.95787598
 0.20520229 0.123301   0.64143809  0.68485511  0.29509309  0.65719205
 0.60906626 0.56890899  0.82614517  0.4408421   0.48018921  0.82674918
 0.37076319 0.56239926  0.47655734  0.16489053  0.79742579  0.57731408
 0.62065454 0.70110719  0.61330581  0.84084355  0.7967645   0.84148374
 0.04915798 0.31650656  0.64326928  0.20878387  0.29682973  0.34488916
 0.54626642 0.35608015  0.37950982  0.42281464  0.62984657  0.46538511
 0.84092615 0.38056331  0.21669412  0.44118415  0.65537459  0.2136067
 0.72679706 0.22969462]
[1]
```

- **单层神经网络**

为了使结构简洁明了，我们使用单层神经网络来预测输出：

```
a = np.matmul(X, theta)
YHat = sigmoid(a)
```

使用 MAML 寻找最优参数值 $\theta$，以在任务间泛化。因此，对于新的任务，可以通过更少的梯度步骤在更短的时间内从较少的数据点中学习。

- **使用 MAML 训练**

现在，定义一个名为 `MAML` 的类，在这个类中实现 MAML 算法。在`__init__`方法中，我们会初始化所有必要的变量。然后，定义 `sigmoid` 激活函数。接下来，定义 `train` 函数。

定义 `MAML` 类以实现 MAML 算法：

```
class MAML(object):
```

定义`__init__`方法并初始化所有必要的变量：

```
def __init__(self):
```

初始化任务数量，也就是每批任务中需要的任务数量：

```
self.num_tasks = 10
```

下面是每个任务中需要的样本点的数量（$k$）：

```
self.num_samples = 10
```

下面是训练轮数，即迭代次数：

```
self.epochs = 1000
```

下面是内循环（内部梯度更新）的超参数：

```
self.alpha = 0.0001
```

下面是外循环（外部梯度更新）的超参数，即元优化：

```
self.beta = 0.0001
```

然后随机初始化模型参数$\theta$：

```
self.theta = np.random.normal(size=50).reshape(50, 1)
```

定义 sigmoid 激活函数：

```
def sigmoid(self,a):
    return 1.0 / (1 + np.exp(-a))
```

开始训练：

```
def train(self):
```

对于每轮训练：

```
for e in range(self.epochs):

    self.theta_ = []
```

对于每批任务中的任务 i：

```
for i in range(self.num_tasks):
```

抽取$k$个数据点，并准备训练集$D_i^{\text{train}}$：

```
XTrain, YTrain = sample_points(self.num_samples)
```

通过单层神经网络预测 YHat 的值：

```
a = np.matmul(XTrain, self.theta)

YHat = self.sigmoid(a)
```

由于在执行分类任务，因此使用交叉熵损失作为损失函数：

```
                loss = ((np.matmul(-YTrain.T, np.log(YHat)) - np.matmul((1
-YTrain.T), np.log(1 - YHat)))/self.num_samples)[0][0]
```

通过计算梯度最小化损失：

```
                gradient = np.matmul(XTrain.T, (YHat - YTrain)) /
self.num_samples
```

更新梯度，得到任务 $T_i$ 的最优参数 $\theta'_t$，使得 $\theta'_t = \theta - \alpha\nabla_\theta L_{T_i}(f_\theta)$：

```
self.theta_.append(self.theta - self.alpha*gradient)
```

初始化元梯度：

```
meta_gradient = np.zeros(self.theta.shape)
```

抽取 $k$ 个数据点，并为元训练准备测试集（元训练集）$D_i^{\text{test}}$：

```
for i in range(self.num_tasks):

    XTest, YTest = sample_points(10)
```

通过单层神经网络预测 YPred 的值：

```
a = np.matmul(XTest, self.theta_[i])
YPred = self.sigmoid(a)
```

计算元梯度：

```
                meta_gradient += np.matmul(XTest.T, (YPred - YTest)) /
self.num_samples
```

使用元梯度更新随机初始化的模型参数 $\theta$：

$$\theta = \theta - \beta\nabla_\theta \sum_{T_i \sim p(T)} L_{T_i}(f_{\theta'_i})$$

```
self.theta = self.theta-self.beta*meta_gradient/self.num_tasks
```

每 1000 轮训练后打印一次损失：

```
if e%1000==0:
    print "Epoch {}: Loss {}\n".format(e,loss)
    print 'Updated Model Parameter Theta\n'
    print 'Sampling Next Batch of Tasks \n'
    print '---------------------------------\n'
```

MAML 类的完整代码如下：

```
class MAML(object):
    def __init__(self):
        #初始化任务数量，即每批任务中所需的任务数量
        self.num_tasks = 10
        #样本点的数量，即每个任务中所需数据点的数量（k）
        self.num_samples = 10

        #训练轮数，即迭代次数
        self.epochs = 10000
        #内循环（内部梯度更新）的超参数
        self.alpha = 0.0001
        #外循环（外部梯度更新）的超参数，即元优化
        self.beta = 0.0001
        #随机初始化模型参数θ
        self.theta = np.random.normal(size=50).reshape(50, 1)
    #定义sigmoid激活函数
    def sigmoid(self,a):
        return 1.0 / (1 + np.exp(-a))
    #开始训练
    def train(self):
        #对于每轮训练
        for e in range(self.epochs):
            self.theta_ = []
            #对于每批任务中的任务i
            for i in range(self.num_tasks):
                #抽取k个数据点，并准备训练集
                XTrain, YTrain = sample_points(self.num_samples)
                a = np.matmul(XTrain, self.theta)

                YHat = self.sigmoid(a)

                #由于在执行分类任务，因此使用交叉熵损失作为损失函数
                loss = ((np.matmul(-YTrain.T, np.log(YHat)) - np.matmul((1
-YTrain.T), np.log(1 - YHat)))/self.num_samples)[0][0]
                #通过计算梯度最小化损失
                gradient = np.matmul(XTrain.T, (YHat - YTrain)) /
self.num_samples

                #更新梯度，得到任务Ti的最优参数θi
                self.theta_.append(self.theta - self.alpha*gradient)
            #初始化元梯度
            meta_gradient = np.zeros(self.theta.shape)
            for i in range(self.num_tasks):
                #抽取k个数据点，并为元训练准备测试集（元训练集）
                XTest, YTest = sample_points(10)

                #预测y的值
                a = np.matmul(XTest, self.theta_[i])
                YPred = self.sigmoid(a)
                #计算元梯度
                meta_gradient += np.matmul(XTest.T, (YPred - YTest)) /
self.num_samples

            #使用元梯度更新随机初始化的模型参数θ
            self.theta = self.theta-self.beta*meta_gradient/self.num_tasks
            if e%1000==0:
```

6

```
            print "Epoch {}: Loss {}\n".format(e,loss)
            print 'Updated Model Parameter Theta\n'
            print 'Sampling Next Batch of Tasks \n'
            print '---------------------------------\n'
```

现在，创建 MAML 类的实例：

```
model = MAML()
```

开始训练模型：

```
model.train()
```

输出如下，可以看到损失从第 0 轮的 2.71 急剧下降到第 3000 轮的 0.5：

```
Epoch 0: Loss 2.71883405043

Updated Model Parameter Theta

Sampling Next Batch of Tasks

---------------------------------

Epoch 1000: Loss 1.7829716017

Updated Model Parameter Theta

Sampling Next Batch of Tasks

---------------------------------

Epoch 2000: Loss 1.29532754055

Updated Model Parameter Theta

Sampling Next Batch of Tasks

---------------------------------

Epoch 3000: Loss 0.599713728648

Updated Model Parameter Theta

Sampling Next Batch of Tasks

---------------------------------
```

### 6.1.3　强化学习中的 MAML

如何将 MAML 应用于**强化学习**场景中呢？在强化学习中，我们的目标是找到正确的策略函数，它将告诉我们在每种状态下执行什么操作。但如何在强化学习中应用元学习呢？假设我们训练智能体（agent）来解决双臂老虎机（two-armed bandit）问题，然而，不能用同一个智能体来

解决四臂老虎机（four-armed bandit）的问题。为了解决这个新的四臂老虎机问题，我们不得不重新训练这个智能体。同样，当出现另一个 $n$ 臂老虎机（$n$-armed bandit）时，即使它与智能体已经学会解决的问题密切相关，我们还是得不断地从头开始训练智能体，以解决新的问题。显然，我们无须这样做。我们可以应用元学习，对智能体进行一组相关任务的训练，使得智能体能够利用其以前的知识，在最短的时间内学习新的相关任务，而不必从头开始训练。

在强化学习中，我们将包含一系列观察（observation）和行为（action）的元组（tuple）称为轨迹（trajectory），因此，在这些轨迹上训练模型来学习最优策略，但应该用哪种算法来训练模型呢？对于 MAML，可以使用任何可以通过梯度下降训练的强化学习算法。我们使用策略梯度（policy gradients）来训练模型。策略梯度通过直接将策略 $\pi$ 的参数 $\pi_\theta$ 参数化为 $\theta$ 来找到最优策略。使用 MAML，我们试图找到这个可在任务间泛化的最优参数 $\theta$ 。

可应选择什么损失函数呢？在强化学习中，我们的目标是通过最大化正回报和最小化负回报来找到最优策略，因此将损失函数设为最小化负回报，即

$$L_{T_i}(f_\theta) = -\mathbb{E}_{x_t, a_t \sim f_\theta, qT_i}[\sum_{t=1}^{H} R_i(x_t, a_t)]$$

上述等式的解释如下：$R_i(x_t, a_t)$ 表示在 $t$ 时刻对状态 $x$ 采取 $a$ 行为的回报，$t = 1 \sim H$ 表示时间步数，其中 $H$ 为上限，即最终时间。

假设有一个由 $\theta$ 影响的模型 $f_\theta()$ 以及任务的分布 $p(T)$ 。首先，用一些随机值初始化参数 $\theta$ 。接下来，从任务的分布中抽取一批任务 $T_i$ （即 $T_i \sim p(T)$ ）。

然后，对每个任务，抽取 $k$ 个轨迹，并构建训练集与测试集：$D_i^{\text{train}}, D_i^{\text{test}} \sim T_i$ 。数据集基本上包含了轨迹信息，比如观察和行为。通过执行梯度下降并找到最优参数 $\theta'$ 来最小化训练集 $D_i^{\text{train}}$ 上的损失：

$$\theta_i' = \theta - \alpha \nabla_\theta L_{T_i}(f_\theta)$$

现在，在抽取下一批任务前，执行一个元更新，也就是说，通过计算相对于最优参数 $\theta_i'$ 的损失梯度来最小化测试集 $D_i^{\text{test}}$ 上的损失，以更新我们的随机初始化参数 $\theta$ ：

$$\theta = \theta - \beta \nabla_\theta \sum_{T_i \sim p(T)} L_{T_i}(f_{\theta_i'})$$

## 6.2 ADML

我们已了解了如何使用 MAML 来找到最优参数 $\theta$ ，该参数可在任务间泛化。下面，我们将学习 MAML 的变体——ADML，它使用干净样本（clean sample）和对抗样本（adversarial sample）

来找到更好、更稳健的初始模型参数 $\theta$ 。让我们先了解一下什么是对抗样本，它产生于对抗攻击（adversarial attack）。假设有一个图像，对抗攻击以一种肉眼无法察觉的方式对图像进行轻微修改，这种修改后的图像称为对抗图像。当我们把这个对抗图像输入模型时，它不能正确地分类。现在有几种用于获得对抗样本的对抗攻击。我们将看到一个常用的方法——**快速梯度符号法**（fast gradient sign method，FGSM）。

## 6.2.1　FGSM

假设我们正在执行图像分类任务。通常，我们通过计算损失来训练模型，并试图通过计算相对于模型参数（如权重）的损失梯度，来最小化损失并更新模型参数。为了得到图像的对抗样本，我们计算相对于图像输入像素（而不是模型参数）的损失梯度。因此，图像的对抗样本基本上就是相对于图像的损失梯度。只需要一个梯度步骤，因此它在计算上是有效的。在计算梯度后，取它的 sign 值。

对抗样本计算如下：

$$X_{\text{adv}} = x + \epsilon \text{sign}(\nabla_x J(x, y_{\text{true}}))$$

参数解释如下：

- $x_{\text{adv}}$ 是对抗图像；
- $x$ 是输入图像；
- $\nabla_x J(x, y_{\text{true}})$ 是相对于输入图像的损失梯度。

如图 6-5 所示，有输入图像 $x$ ，通过在实际图像上添加图像损失梯度的 sign 值来得到对抗图像。

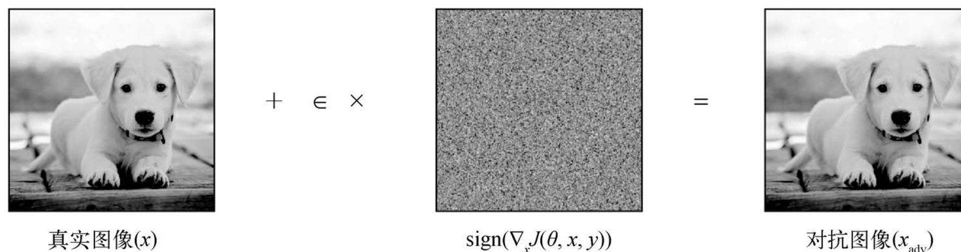



图　6-5

## 6.2.2　ADML

我们现在已了解了什么是对抗样本以及如何生成对抗样本，接下来将学习如何在元学习中使用这些对抗样本。我们用干净样本和对抗样本来训练元学习模型，但为什么要使用对抗样本训练

模型呢？因为它能帮助我们找到稳健的模型参数 $\theta$。在算法的内循环和外循环中均使用了干净样本和对抗样本，它们对更新模型参数的贡献相同。ADML 利用干净样本和对抗样本之间的这种不断变化的相关性来获得更好、更稳健的模型初始化参数，从而使我们的参数在对抗样本中具有稳健性，并能很好地在新任务中泛化。

假设有任务的分布 $p(T)$，从任务的分布中抽取一批任务 $T_i$。对于每个任务，抽取 $k$ 个数据点，并构建训练集与测试集。

在 ADML 中，我们同时从干净样本与对抗样本中抽取训练集与测试集，即 $D_{\text{clean}_i}^{\text{train}}$、$D_{\text{adv}_i}^{\text{train}}$、$D_{\text{clean}_i}^{\text{test}}$ 和 $D_{\text{adv}_i}^{\text{test}}$。

现在在训练集上计算损失，通过梯度下降最小化损失，并找到最优参数 $\theta'$。由于有干净训练集和对抗训练集，我们对这两个集合进行梯度下降，分别找到干净训练集和对抗训练集的最优参数 $\theta'_{\text{clean}_i}$ 和 $\theta'_{\text{adv}_i}$：

$$\theta'_{\text{clean}_i} = \theta - \alpha_1 \nabla_\theta L_{T_i}(f_\theta, D_{\text{clean}_i}^{\text{train}})$$
$$\theta'_{\text{adv}_i} = \theta - \alpha_2 \nabla_\theta L_{T_i}(f_\theta, D_{\text{adv}_i}^{\text{train}})$$

现在进行元训练。通过计算相对于上一步中最优参数 $\theta'$ 的损失的梯度，在测试集上最小化损失，来找到最优参数 $\theta$。

通过计算相对于最优参数 $\theta'_{\text{clean}_i}$ 和 $\theta'_{\text{adv}_i}$ 的损失的梯度，在干净训练集 $D_{\text{clean}_i}^{\text{test}}$ 和对抗训练集 $D_{\text{adv}_i}^{\text{test}}$ 上最小化损失，来更新模型参数 $\theta$：

$$\theta = \theta - \beta_1 \nabla_\theta \sum_{T_i \sim p(T)} L_{T_i}(f_{\theta'_{\text{clean}_i}}, D_{\text{clean}_i}^{\text{test}})$$
$$\theta = \theta - \beta_2 \nabla_\theta \sum_{T_i \sim p(T)} L_{T_i}(f_{\theta'_{\text{adv}_i}}, D_{\text{adv}_i}^{\text{test}})$$

### 6.2.3 从头构建 ADML

在上一节，我们了解了 ADML 的工作原理，以及如何同时使用干净样本和对抗样本训练模型，以获得一个更好、更稳健的模型参数 $\theta$，用于在任务间泛化。这一节我们将通过从头开始编写代码来更好地理解 ADML，并将考虑一个简单的二分类任务。我们随机生成输入数据，并用单层神经网络训练它，试图找到最优参数 $\theta$（`theta`）。下面我们将逐步了解 ADML 究竟是如何运作的。

你也可以在 GitHub 网站搜索 Hands-On-Meta-Learning-With-Python，在带有注释的 Jupyter Notebook 中查看相应代码。

首先，导入所有必要的库：

```
import tensorflow as tf
import numpy as np
```

### 1. 生成数据点

现在定义一个名为 sample_points 的函数来生成干净输入对(x, y)。它以 k 作为输入参数，即抽取(x, y)对的数量：

```
def sample_points(k):
    x = np.random.rand(k,50)
    y = np.random.choice([0, 1], size=k, p=[.5, .5]).reshape([-1,1])
    return x,y
```

上述函数的输出如下：

```
x, y = sample_points(10)
print x[0]
print y[0]

[0.69922136 0.77305793 0.72227583 0.45291578 0.52828294 0.65308614
 0.77281836 0.59878078 0.71554901 0.51660327 0.65538137 0.25267594
 0.13763862 0.12522582 0.16336571 0.87987815 0.64465771 0.86281232
 0.24503599 0.85324859 0.62247917 0.58166159 0.47871545 0.75025566
 0.87919612 0.49545388 0.31058753 0.66306459 0.34621453 0.56970739
 0.84310111 0.08747573 0.48944231 0.50061581 0.86215915 0.3248433
 0.01350084 0.23846395 0.91015074 0.04968178 0.59098773 0.74692099
 0.92763503 0.16319537 0.69655162 0.20419323 0.58241944 0.15703596
 0.76047838 0.93452557]
[0]
```

### 2. FGSM

下面定义一个名为 FGSM 的函数来生成对抗输入，并使用 FGSM 来生成对抗样本。我们已看到 FGSM 如何通过计算相对于输入而不是模型参数的梯度来生成对抗样本对。因此，将干净样本对(x,y)作为输入，并生成对抗样本对(x_adv,y)：

```
def FGSM(x,y):

    #输入 x 与 y 的占位符
    X = tf.placeholder(tf.float32)
    Y = tf.placeholder(tf.float32)

    #使用随机值初始化 theta
    theta = tf.Variable(tf.zeros([50,1]))

    #预测 y 的值
    YHat = tf.nn.softmax(tf.matmul(X, theta))

    #计算损失
```

```
    loss = tf.reduce_mean(-tf.reduce_sum(Y*tf.log(YHat),
reduction_indices=1))

    #现在计算相对于输入 x 而不是模型参数 theta 的损失函数的梯度
    gradient = ((tf.gradients(loss,X)[0]))
    #计算对抗输入
    #即 x_adv = x + epsilon = sign ( nabla_x J(X, Y))
    X_adv = X + 0.2*tf.sign(gradient)
    X_adv = tf.clip_by_value(X_adv,-1.0,1.0)

    #启动 TensorFlow 会话
    with tf.Session() as sess:

        sess.run(tf.global_variables_initializer())
        X_adv = sess.run(X_adv, feed_dict={X: x, Y: y})
    return X_adv, y
```

### 3. 单层神经网络

我们使用单层神经网络预测输出：

```
a = np.matmul(X, theta)
 YHat = sigmoid(a)
```

使用 ADML 寻找最优参数值$\theta$，以在任务间泛化。因此，对于新任务，可以通过更少的梯度步骤在更短的时间内从少量数据点中学习。

### 4. ADML

现在，定义一个名为 ADML 的类，并在其中实现 ADML 算法。在__init__方法中，我们将初始化所有必要的变量，然后定义 sigmoid 函数和 train 函数。

我们会逐步阐述，并在最后提供完整代码：

```
class ADML(object):
```

定义__init__方法，并初始化必要的变量：

```
def __init__(self):
```

初始化任务的数量，即每批任务中我们需要的任务数量：

```
self.num_tasks = 2
```

初始化样本的数量，即每个任务中需要的数据点数量（$k$）：

```
self.num_samples = 10
```

初始化轮数，即训练迭代次数：

```
self.epochs = 100
```

内循环（内部梯度更新）的超参数如下：

```
#干净样本

self.alpha1 = 0.0001

#对抗样本

self.alpha2 = 0.0001
```

外循环（外部梯度更新，即元优化）的超参数如下：

```
#干净样本
self.beta1 = 0.0001
#对抗样本
self.beta2 = 0.0001
```

随机初始化模型参数 `theta`：

```
self.theta = np.random.normal(size=50).reshape(50, 1)
```

定义 `sigmoid` 激活函数：

```
def sigmoid(self,a):
    return 1.0 / (1 + np.exp(-a))
```

下面来看如何训练网络：

```
def train(self):
```

对于每轮训练：

```
for e in range(self.epochs):

    #干净样本的theta'
    self.theta_clean = []

    #对抗样本的theta'
    self.theta_adv = []
```

对于每批任务中的任务 `i`：

```
for i in range(self.num_tasks):
```

抽取 $k$ 个数据点，并准备训练集。首先，抽取干净数据点，即 $D_{\text{clean}_i}^{\text{train}}$：

```
XTrain_clean, YTrain_clean = sample_points(self.num_samples)
```

将干净样本输入 FGSM，得到对抗样本 $D_{\text{adv}_i}^{\text{train}}$：

```
XTrain_adv, YTrain_adv = FGSM(XTrain_clean,YTrain_clean)
```

现在，计算 $\theta'_{\text{clean}_i}$，并将其存入 `theta_clean`。使用单层神经网络预测输出 `y`：

```
a = np.matmul(XTrain_clean, self.theta)

YHat = self.sigmoid(a)
```

由于在执行分类，我们使用交叉熵损失作为损失函数：

```
                loss = ((np.matmul(-YTrain_clean.T, np.log(YHat)) -
np.matmul((1 -YTrain_clean.T), np.log(1 - YHat)))/self.num_samples)[0][0]
```

通过计算梯度来最小化损失：

```
                gradient = np.matmul(XTrain_clean.T, (YHat - YTrain_clean))
/ self.num_samples
```

更新梯度并寻找干净样本的最优参数 $\theta'_{\text{clean}_i}$ ， $\theta'_{\text{clean}_i} = \theta - \alpha_1 \nabla_\theta L_{T_i}(f_\theta, D^{\text{train}}_{\text{clean}_i})$ ：

```
self.theta_clean.append(self.theta - self.alpha1*gradient)
```

现在计算对抗样本的 $\theta'_{\text{adv}_i}$ ，并将其存入 theta_adv：

```
                #预测输出 y
                a = (np.matmul(XTrain_adv, self.theta))

                YHat = self.sigmoid(a)

                #计算交叉熵损失
                loss = ((np.matmul(-YTrain_adv.T, np.log(YHat)) -
np.matmul((1 -YTrain_adv.T), np.log(1 - YHat)))/self.num_samples)[0][0]
                #通过计算梯度最小化损失
                gradient = np.matmul(XTrain_adv.T, (YHat - YTrain_adv)) /
self.num_samples
```

更新梯度并寻找干净样本的最优参数 $\theta'_{\text{adv}_i}$ ， $\theta'_{\text{adv}_i} = \theta - \alpha_2 \nabla_\theta L_{T_i}(f_\theta, D^{\text{train}}_{\text{adv}_i})$ ：

```
self.theta_adv.append(self.theta - self.alpha2*gradient)
```

为干净样本与对抗样本初始化元梯度：

```
meta_gradient_clean = np.zeros(self.theta.shape)

#为对抗样本初始化元梯度
meta_gradient_adv = np.zeros(self.theta.shape)
```

对于每一批任务中的任务 i：

```
for i in range(self.num_tasks):
```

抽取 $k$ 个数据点，并为元训练准备干净测试集与对抗测试集（元训练集），即 $D^{\text{test}}_{\text{clean}_i}$ 和 $D^{\text{test}}_{\text{adv}_i}$ ：

```
#首先，抽取干净样本点
XTest_clean, YTest_clean = sample_points(self.num_samples)

#将干净样本输入 FGSM，得到对抗样本
XTest_adv, YTest_adv = sample_points(self.num_samples)
```

6

首先，计算干净样本的元梯度：

```
                #预测y的值
                a = np.matmul(XTest_clean, self.theta_clean[i])
                YPred = self.sigmoid(a)
                #计算元梯度
                meta_gradient_clean += np.matmul(XTest_clean.T, (YPred -
YTest_clean)) / self.num_samples
```

现在，计算对抗样本的元梯度：

```
                #预测y的值
                a = (np.matmul(XTest_adv, self.theta_adv[i]))
                YPred = self.sigmoid(a)
                #计算元梯度
                meta_gradient_adv += np.matmul(XTest_adv.T, (YPred -
YTest_adv)) / self.num_samples
```

使用干净样本与对抗样本的元梯度，更新随机初始化的模型参数$\theta$：

$$\theta = \theta - \beta_1 \nabla_\theta \sum_{T_i \sim p(T)} L_{T_i}(f_{\theta'_{\text{clean}_i}}, D_{\text{clean}_i}^{\text{test}})$$

$$\theta = \theta - \beta_2 \nabla_\theta \sum_{T_i \sim p(T)} L_{T_i}(f_{\theta'_{\text{adv}_i}}, D_{\text{adv}_i}^{\text{test}})$$

```
            self.theta = self.theta-
self.beta1*meta_gradient_clean/self.num_tasks

            self.theta = self.theta-
self.beta2*meta_gradient_adv/self.num_tasks
```

每 10 轮打印一次损失：

```
if e%10==0:
    print "Epoch {}: Loss {}\n".format(e,loss)
    print 'Updated Model Parameter Theta\n'
    print 'Sampling Next Batch of Tasks \n'
    print '---------------------------------\n'
```

完整的 ADML 类如下：

```
class ADML(object):
    def __init__(self):

        #初始化任务的数量，即每批任务中我们需要的任务数量
        self.num_tasks = 2
        #样本的数量，即每个任务中我们需要的数据点数量（k）
        self.num_samples = 10

        #轮数，即训练迭代次数
        self.epochs = 100
        #内循环（内部梯度更新）的超参数
```

```
        #干净样本
        self.alpha1 = 0.0001

        #对抗样本
        self.alpha2 = 0.0001
        #外循环（外部梯度更新，即元优化）的超参数
        #干净样本
        self.beta1 = 0.0001
        #对抗样本
        self.beta2 = 0.0001

        #随机初始化模型参数 theta
        self.theta = np.random.normal(size=50).reshape(50, 1)

    #定义 sigmoid 激活函数
    def sigmoid(self,a):
        return 1.0 / (1 + np.exp(-a))
    #下面来看如何训练网络
    def train(self):
        #对于每轮训练
        for e in range(self.epochs):
            #干净样本的 theta'
            self.theta_clean = []

            #对抗样本的 theta'
            self.theta_adv = []
            #对于每批任务中的任务 i
            for i in range(self.num_tasks):
                #抽取 k 个数据点，并准备训练集

                #首先，抽取干净数据点
                XTrain_clean, YTrain_clean = sample_points(self.num_samples)

                #将干净样本输入 FGSM，得到对抗样本
                XTrain_adv, YTrain_adv = FGSM(XTrain_clean,YTrain_clean)

                #1.首先，计算干净样本的 theta'，并将其存入 theta_clean
                #预测输出 y
                a = np.matmul(XTrain_clean, self.theta)

                YHat = self.sigmoid(a)

                #由于在执行分类，因此使用交叉熵损失作为损失函数
                loss = ((np.matmul(-YTrain_clean.T, np.log(YHat)) -
np.matmul((1 -YTrain_clean.T), np.log(1 - YHat)))/self.num_samples)[0][0]
                #通过计算梯度来最小化损失
                gradient = np.matmul(XTrain_clean.T, (YHat - YTrain_clean))
/ self.num_samples

                #更新梯度并寻找干净样本的最优参数 theta'
                self.theta_clean.append(self.theta - self.alpha1*gradient)
                #2. 现在，计算对抗样本的 theta'，并将其存入 theta_clean

                #预测输出 y
```

```
                a = (np.matmul(XTrain_adv, self.theta))

                YHat = self.sigmoid(a)

                #计算交叉熵损失
                loss = ((np.matmul(-YTrain_adv.T, np.log(YHat)) -
np.matmul((1 -YTrain_adv.T), np.log(1 - YHat)))/self.num_samples)[0][0]
                #通过计算梯度最小化损失
                gradient = np.matmul(XTrain_adv.T, (YHat - YTrain_adv)) /
self.num_samples

                #更新梯度并寻找对抗样本的最优参数 theta'
                self.theta_adv.append(self.theta - self.alpha2*gradient)
            #为干净样本初始化元梯度
            meta_gradient_clean = np.zeros(self.theta.shape)

            #为对抗样本初始化元梯度
            meta_gradient_adv = np.zeros(self.theta.shape)
            for i in range(self.num_tasks):
                #抽取 k 个数据点，并为元训练准备测试集

                #首先，抽取干净样本点
                XTest_clean, YTest_clean = sample_points(self.num_samples)

                #将干净样本输入 FGSM，得到对抗样本
                XTest_adv, YTest_adv = sample_points(self.num_samples)
                #1.首先，计算干净样本的元梯度

                #预测 y 的值
                a = np.matmul(XTest_clean, self.theta_clean[i])
                YPred = self.sigmoid(a)
                #计算元梯度
                meta_gradient_clean += np.matmul(XTest_clean.T, (YPred -
YTest_clean)) / self.num_samples

                #2.现在，计算对抗样本的元梯度
                #预测 y 的值
                a = (np.matmul(XTest_adv, self.theta_adv[i]))
                YPred = self.sigmoid(a)
                #计算元梯度
                meta_gradient_adv += np.matmul(XTest_adv.T, (YPred -
YTest_adv)) / self.num_samples

            #使用干净样本与对抗样本的元梯度，更新随机初始化的模型参数 theta
            self.theta = self.theta-
self.beta1*meta_gradient_clean/self.num_tasks

            self.theta = self.theta-
self.beta2*meta_gradient_adv/self.num_tasks
            if e%10==0:
                print "Epoch {}: Loss {}\n".format(e,loss)
                print 'Updated Model Parameter Theta\n'
                print 'Sampling Next Batch of Tasks \n'
                print '---------------------------------\n'
```

创建 ADML 类的实例：

```
model = ADML()
```

然后，开始训练模型：

```
model.train()
```

注意损失是如何随着训练轮数的增加而降低的：

```
Epoch 0: Loss 100.25943711532

Updated Model Parameter Theta

Sampling Next Batch of Tasks

---------------------------------

Epoch 10: Loss 2.13533264312

Updated Model Parameter Theta

Sampling Next Batch of Tasks

---------------------------------

Epoch 20: Loss 0.426824910313

Updated Model Parameter Theta

Sampling Next Batch of Tasks
```

## 6.3 CAML



我们已看到了 MAML 如何找到模型的最优初始参数，这使得它可以很容易地适应梯度步骤较少的新任务。下面将学习一个有趣的 MAML 变体——CAML。CAML 的概念很简单，和 MAML 一样，它也试图找到更好的初始参数。我们了解了 MAML 如何使用两个循环来进行学习：在内循环中，它学习特定于任务的参数并试图使用梯度下降最小化损失；在外循环中，它更新模型参数，以减少几个任务间的期望损失，这使得我们可以将更新后的模型参数作为相关任务更优的初始值。

在 CAML 中稍微调整了 MAML 算法。这里没有使用单个模型参数，而是将模型参数一分为二。

- **上下文参数**（context parameter）：是在内循环中更新的特定于任务的参数 $\phi$。它特定于具体的任务，代表单个任务的嵌入。
- **共享参数**（shared parameter）：记为 $\theta$，在任务间共享，并在外循环中更新，以找到最优的模型参数。

因此，上下文参数在内循环中根据任务而变化，而共享参数在任务之间共享并用于外循环中的元训练。在每个适应（adaption）步骤前，我们将上下文参数初始化为零。

但把参数一分为二有什么用呢？这避免对特定任务的过度拟合，加速学习，并提高存储器使用效率。

### CAML 算法

下面让我们逐步了解 CAML 的工作原理。

(1) 假设有一个由 $\theta$ 影响的模型 $f$ 以及任务的分布 $p(T)$。首先，随机初始化参数 $\theta$，并初始化上下文参数 $\phi_0 = 0$。

(2) 现在，从任务的分布中抽取一批任务 $T_i$，即 $T_i \sim p(T)$。

(3) **内循环**：对于任务 $T$ 中的每个任务 $T_i$，我们抽取 $k$ 个数据点并准备训练集与测试集：

$$D_i^{\text{train}} = \{(x_1, y_1), (x_2, y_2), \cdots, (x_k, y_k)\}$$
$$D_i^{\text{test}} = \{(x_1, y_1), (x_2, y_2), \cdots, (x_k, y_k)\}$$

现在，将上下文参数设为 0：

$$\phi_0 = 0$$

然后，在 $D_i^{\text{train}}$ 上计算损失，利用梯度下降最小化损失，得到特定于任务的参数 $\phi_i$：

$$\phi_i = \phi_0 - \alpha \nabla_\phi L_{T_i}(f_{\phi_0, \theta})$$

(4) **外循环**：现在，我们在测试集中执行元优化。这里，我们试图将测试集 $D_i^{test}$ 中的损失最小化，并找到最优参数：

$$\theta = \theta - \beta \nabla_\phi \sum_{T_i \sim p(T)} L_{T_i}(f_{\phi_i, \theta})$$

(5) 对步骤(2)~(4)进行 $n$ 次迭代。

## 6.4　小结

在本章，我们已学会了如何找到最优模型参数 $\theta$，以在任务间泛化，这使得我们可以减少梯度步骤并快速学习新的相关任务。从 MAML 开始，我们了解了 MAML 如何执行元优化来计算最优模型参数；接下来，掌握了对抗性元学习，并使用干净样本和对抗样本来寻找稳健的初始模型参数；之后，学习了 CAML，并了解了如何使用两种不同的参数——一种用于在任务内学习，另一种用于更新模型参数。

在下一章，我们将学习 Meta-SGD 和 Reptile 算法，该算法同样用于寻找更好的模型初始参数。

## 6.5 思考题

(1) 什么是 NTM？
(2) 为什么 MAML 是模型无关的？
(3) 什么是对抗元学习？
(4) 什么是 FGSM？
(5) 什么是上下文参数？
(6) 什么是共享参数？

## 6.6 延伸阅读

- MAML 论文：Chelsea Finn、Pieter Abbeel 和 Sergey Levine 的文章 *Model-Agnostic Meta-Learning for Fast Adaptation of Deep Networks*。
- 对抗元学习论文：Chengxiang Yin、Jian Tang、Zhiyuan Xu 等人的文章 *Adversarial Meta-Learning*。
- CAML 论文：Luisa M Zintgraf、Kyriacos Shiarlis、Vitaly Kurin 等人的文章 *Fast Context Adaptation via Meta-Learning*。

第 7 章

# Meta-SGD 和 Reptile

在上一章，我们学习了如何使用 MAML 来寻找可在任务间泛化的最优参数，及其如何通过计算元梯度和执行元优化来计算这个最优参数；了解了对抗元学习，它通过添加对抗样本来增强 MAML，并允许 MAML 在干净样本和对抗样本之间进行比较，以找到最佳参数；阐释了 CAML，即元学习的上下文适应。在本章，我们将学习 Meta-SGD——另一种用于快速学习的元学习算法。与 MAML 不同，Meta-SGD 不仅能找到最优的参数，还能找到最优的学习率和更新方向。我们将了解如何在监督和强化学习场景中使用 Meta-SGD，以及如何从头构建 Meta-SGD；学习 Reptile 算法，它改进了 MAML；阐释 Reptile 与 MAML 的不同之处，以及如何在正弦曲线回归任务中使用 Reptile。

本章内容包括：

- Meta-SGD；
- 监督学习中的 Meta-SGD；
- 强化学习中的 Meta-SGD；
- 从头构建 Meta-SGD；
- Reptile；
- 使用 Reptile 进行正弦曲线回归。

## 7.1 Meta-SGD

假设有一个任务 $T$。使用由某个参数 $\theta$ 影响的模型 $f$，训练模型使损失最小化。我们使用梯度下降最小化损失，并找出模型的最优参数 $\theta$。

让我们回忆一下梯度下降的更新规则：

$$\theta = \theta - \alpha \nabla_{\theta} L_{T_i}(f_{\theta})$$

以下是梯度下降的核心元素：

- ❑ 参数$\theta$
- ❑ 学习率$\alpha$
- ❑ 更新方向

我们通常将参数$\theta$设置为随机值，并尝试在训练过程中找到最优值。将学习率$\alpha$的值设置为小数值，或者使其随着时间的推移而衰减，同时依照梯度设置更新方向。是否可以通过元学习来学习所有的这些梯度下降关键元素，使得我们可以从少量数据点中快速学习？在上一章，我们已经看到了 MAML 如何找到可在任务间泛化的最优初始参数$\theta$。在初始参数最优的情况下，我们可以采取更少的梯度步骤，快速学习新的任务。

那么，现在可以学习可在任务间泛化的最优的学习率和更新方向，从而实现更快的收敛和训练吗？让我们看看如何通过将 Meta-SGD 与 MAML 进行比较来学习。回想一下，在 MAML 内循环中，通过梯度下降使损失最小化，从而找到每个任务$T_i$的最优参数$\theta_i'$：

$$\theta_i' = \theta - \alpha \nabla_\theta L_{T_i}(f_\theta)$$

对于 Meta-SGD，上述方程可写作：

$$\theta_i' = \theta - \boldsymbol{\alpha} \circ \nabla_\theta L_{T_i}(f_\theta)$$

这有什么区别呢？这里$\boldsymbol{\alpha}$不仅仅是小的标量值，而且是向量。我们将随机初始化$\boldsymbol{\alpha}$，使其具有与$\theta$相同的形状。我们将$\theta$作为初始参数，并将$\boldsymbol{\alpha} \circ \nabla_\theta L_{T_i}(f_\theta)$作为自适应项。因此，自适应项$\boldsymbol{\alpha} \circ \nabla_\theta L_{T_i}(f_\theta)$代表更新方向，其长度为学习率。在$\boldsymbol{\alpha} \circ \nabla_\theta L_{T_i}(f_\theta)$方向上（而不是梯度方向$\nabla_\theta L_{T_i}(f_\theta)$上）更新值，学习率暗含在自适应项中。

因此，在 Meta-SGD 中，不是使用小的标量值来初始化学习率$\boldsymbol{\alpha}$，而是使用与$\theta$形状相同的随机值初始化学习率，并与$\theta$一同接受学习。抽样一批任务，对于每个任务，抽取$k$个数据点，并使用梯度下降最小化损失，但更新方程变成下列形式：

$$\theta_i' = \theta - \boldsymbol{\alpha} \circ \nabla_\theta L_{T_i}(f_\theta)$$

也就是说，我们的更新方向是自适应项方向，而不是梯度方向，并同时学习$\theta$与$\boldsymbol{\alpha}$。

现在，在外循环中，我们执行元优化——也就是说，计算相对于最优参数$\theta_i'$的损失梯度，并更新随机初始化的模型参数$\theta$。在 Meta-SGD 中，不单独更新$\theta$，而是同时更新随机初始化的$\boldsymbol{\alpha}$：

$$\theta = \theta - \beta \nabla_\theta \sum_{T_i \sim p(T)} L_{T_i}(f_{\theta_i'})$$

$$\boldsymbol{\alpha} = \boldsymbol{\alpha} - \beta \nabla_{\boldsymbol{\alpha}} \sum_{T_i \sim p(T)} L_{T_i}(f_{\theta_i'})$$

如你所见，Meta-SGD 只是对 MAML 的一个小小的调整。在 MAML 中，我们随机初始化模型参数 $\theta$，并尝试找到可在任务间泛化的最优参数。在 Meta-SGD 中，我们不仅学习模型参数 $\theta$，还学习暗含于自适应项中的学习率和更新方向。

### 7.1.1　监督学习中的 Meta-SGD

现在，我们将看到如何在监督学习场景中使用 Meta-SGD。与 MAML 一样，我们可以将 Meta-SGD 应用于任何监督学习问题，无论是回归还是分类，都可以通过梯度下降进行训练。首先，我们需要定义希望使用的损失函数。例如，如果要分类，可以使用交叉熵作为损失函数；如果要回归，可以使用均方误差作为损失函数。可以使用任何适合任务的损失函数。让我们一步一步来。

(1) 假设有一个由 $\theta$ 影响的模型 $f$ 以及任务的分布 $p(T)$。首先，随机初始化模型参数 $\theta$，并将 $\boldsymbol{\alpha}$ 初始化为与 $\theta$ 相同的形状。

(2) 从任务的分布中抽取一批任务 $T_i$，即 $T_i \sim p(T)$。假设抽取了 3 个任务 $T = \{T_1, T_2, T_3\}$。

(3) **内循环**：对于任务 $T$ 中的每个任务 $T_i$，抽取 $k$ 个数据点并准备训练集与测试集：

$$D_i^{\text{train}} = \{(x_1, y_1), (x_2, y_2), \cdots, (x_k, y_k)\}$$
$$D_i^{\text{test}} = \{(x_1, y_1), (x_2, y_2), \cdots, (x_k, y_k)\}$$

现在，对 $D_i^{\text{train}}$ 使用某种监督学习算法，计算损失，并利用梯度下降最小化损失，得到最优参数 $\theta_i'$：

$$\theta_i' = \theta - \boldsymbol{\alpha} \circ \nabla_\theta L_{T_i}(f_\theta)$$

对于每一个任务，抽取 $k$ 个数据点，最小化训练集 $D_i^{\text{train}}$ 上的损失，并得到最优参数 $\theta_i'$。当抽取 3 个任务时，我们将有 3 个最优参数 $\{\theta_1', \theta_2', \theta_3'\}$。

(4) **外循环**：现在，在测试集（元训练集）中执行元优化，即，在这里，我们试图将测试集 $D_i^{\text{test}}$ 中的损失最小化。我们通过计算相对于上一步最优参数 $\theta_i'$ 的梯度以减少损失。我们不仅更新 $\theta$，还更新随机初始化的参数 $\boldsymbol{\alpha}$，如下所示：

$$\theta = \theta - \beta \nabla_\theta \sum_{T_i \sim p(T)} L_{T_i}(f_{\theta_i'})$$
$$\boldsymbol{\alpha} = \boldsymbol{\alpha} - \beta \nabla_{\boldsymbol{\alpha}} \sum_{T_i \sim p(T)} L_{T_i}(f_{\theta_i'})$$

(5) 对步骤(2)~(4)进行 $n$ 次迭代。

### 从头构建 Meta-SGD

在上一节，我们了解了 Meta-SGD 是如何工作的，以及 Meta-SGD 如何获得一个更好、更稳健的模型参数$\theta$，该参数可在任务间泛化，并具有最佳的学习率和更新方向。现在，为了更好地理解，我们将从头开始写代码，并将考虑一个简单的二分类任务，就像在 MAML 中所做的那样。我们随机生成输入数据，并用一个简单的单层神经网络训练它，试图找到最优参数$\theta$。我们将逐步地学习。

你也可以在 GitHub 网站搜索 Hands-On-Meta-Learning-With-Python，在带有注释的 Jupyter Notebook 中查看相应代码。

首先，导入 numpy 库：

```
import numpy as np
```

- **生成数据点**

现在，定义一个名为 sample_points 的函数来生成输入(x, y)对。它以参数 k 为输入，表示抽取的(x, y)对的个数：

```
def sample_points(k):
    x = np.random.rand(k,50)
    y = np.random.choice([0, 1], size=k, p=[.5, .5]).reshape([-1,1])
    return x,y
```

上述函数输出如下：

```
x, y = sample_points(10)
print x[0]
print y[0]

[5.01913307e-01 1.01874941e-01 7.16678998e-01 3.90294047e-01
 2.95330904e-01 8.66751993e-01 5.09988127e-01 8.59389493e-01
 5.16202142e-01 7.92016358e-01 8.24237307e-01 7.76739141e-01
 8.57034917e-01 2.75862141e-01 6.44874856e-01 2.75248940e-01
 5.67665047e-01 9.61564994e-01 7.58931873e-01 1.08989614e-02
 7.69325529e-01 4.05955016e-01 1.98799935e-01 9.94134622e-01
 3.07179216e-01 1.34756367e-01 2.92326855e-01 5.00026528e-01
 7.23673231e-01 5.28698231e-01 1.52495715e-01 9.20139339e-01
 1.76127500e-02 2.42244262e-01 7.09515862e-01 7.10358091e-01
 6.47656449e-01 5.15623266e-01 8.77002211e-01 4.18744855e-01
 9.67902538e-01 8.79261670e-01 5.88524781e-01 5.11397703e-02
 7.07513737e-01 4.61998029e-01 8.77306226e-01 5.32049083e-01
 8.07178697e-01 5.01521846e-04]
[1]
```

- **单层神经网络**

我们使用单层神经网络来预测输出：

```
a = np.matmul(X, theta)
YHat = sigmoid(a)
```

使用 Meta-SGD 来寻找可在任务间泛化的最优参数值 theta、学习率和梯度更新方向。因此，对于新任务，可以通过更少的梯度步骤在更短的时间内从少量数据点中学习。

- **Meta-SGD**

现在，定义一个名为 MetaSGD 的类，在这个类中实现 Meta-SGD 算法。在__init__方法中，我们将初始化所有必要的变量，然后定义 sigmoid 激活函数与 train 函数：

```
class MetaSGD(object):
```

定义__init__方法，并初始化所有必要的变量。

```
def __init__(self):
    #初始化任务的数量，即每批任务中需要的任务数量
    self.num_tasks = 2
    #样本的数量，即每个任务中需要的数据点数量（k）
    self.num_samples = 10

    #轮数，即训练迭代次数
    self.epochs = 10000
    #外循环（外部梯度更新）的超参数，即元优化
    self.beta = 0.0001
    #随机初始化模型参数 theta
    self.theta = np.random.normal(size=50).reshape(50, 1)
    #将 alpha 随机初始化为与 theta 相同的形状
    self.alpha = np.random.normal(size=50).reshape(50, 1)
```

定义 sigmoid 激活函数：

```
def sigmoid(self,a):
    return 1.0 / (1 + np.exp(-a))
```

现在，开始训练：

```
def train(self):
```

对于每轮训练：

```
for e in range(self.epochs):

    self.theta_ = []
```

对于每一批任务中的任务 i：

```
for i in range(self.num_tasks):
```

抽取 $k$ 个数据点，并准备训练集：

```
XTrain, YTrain = sample_points(self.num_samples)
```

然后，使用单层神经网络预测 $y$ 的值：

```
a = np.matmul(XTrain, self.theta)

YHat = self.sigmoid(a)
```

计算损失与梯度：

```
#由于在执行分类，我们使用交叉熵损失作为损失函数
loss = ((np.matmul(-YTrain.T, np.log(YHat)) - np.matmul((1-YTrain.T),
np.log(1 - YHat)))/self.num_samples)[0][0]
#通过计算梯度来最小化损失
gradient = np.matmul(XTrain.T, (YHat - YTrain)) / self.num_samples
```

更新梯度并寻找每个任务的最优参数 $\theta'$：

```
self.theta_.append(self.theta - (np.multiply(self.alpha,gradient)))
```

初始化元梯度：

```
meta_gradient = np.zeros(self.theta.shape)

for i in range(self.num_tasks):
```

抽取 $k$ 个数据点，并为元训练准备训练集 $D_i^{\text{test}}$：

```
XTest, YTest = sample_points(10)
```

预测 $y$ 的值：

```
a = np.matmul(XTest, self.theta_[i])
YPred = self.sigmoid(a)
```

计算元梯度：

```
meta_gradient += np.matmul(XTest.T, (YPred - YTest)) / self.num_samples
```

现在，更新模型参数 `theta` 和 `alpha`：

$$\theta = \theta - \beta\nabla_{\theta} \sum_{T_i \sim p(T)} L_{T_i}(f_{\theta_i'})$$

$$\boldsymbol{\alpha} = \boldsymbol{\alpha} - \beta\nabla_{\boldsymbol{\alpha}} \sum_{T_i \sim p(T)} L_{T_i}(f_{\theta_i'})$$

```
self.theta = self.theta-self.beta*meta_gradient/self.num_tasks
self.alpha = self.alpha-self.beta*meta_gradient/self.num_tasks
```

每 1000 轮打印一次损失：

```
if e%1000==0:
    print "Epoch {}: Loss {}\n".format(e,loss)
    print 'Updated Model Parameter Theta\n'
    print 'Sampling Next Batch of Tasks \n'
    print '---------------------------------\n'
```

MetaSGD 的完整代码如下：

```
class MetaSGD(object):
    def __init__(self):
        #初始化任务的数量，即每批任务中需要的任务数量
        self.num_tasks = 2
        #样本的数量，即每个任务中需要的数据点数量（k）
        self.num_samples = 10

        #轮数，即训练迭代次数
        self.epochs = 10000
        #内循环（内部梯度更新）的超参数
        self.alpha = 0.0001
        #外循环（外部梯度更新）的超参数
        self.beta = 0.0001
        #随机初始化模型参数theta
        self.theta = np.random.normal(size=50).reshape(50, 1)
        #将alpha随机初始化为与theta相同的形状
        self.alpha = np.random.normal(size=50).reshape(50, 1)
    #定义sigmoid激活函数
    def sigmoid(self,a):
        return 1.0 / (1 + np.exp(-a))
    #现在，我们开始训练
    def train(self):
        #对于每轮训练
        for e in range(self.epochs):
            self.theta_ = []
            #对于每一批任务中的任务i
            for i in range(self.num_tasks):
                #抽取k个数据点，并准备训练集
                XTrain, YTrain = sample_points(self.num_samples)
                a = np.matmul(XTrain, self.theta)

                YHat = self.sigmoid(a)

                #由于在执行分类，我们使用交叉熵损失作为损失函数
                loss = ((np.matmul(-YTrain.T, np.log(YHat)) - np.matmul((1
-YTrain.T), np.log(1 - YHat)))/self.num_samples)[0][0]
                #通过计算梯度来使损失最小
                gradient = np.matmul(XTrain.T, (YHat - YTrain)) /
self.num_samples

                #更新梯度并寻找每个任务的最优参数theta'
                self.theta_.append(self.theta -
(np.multiply(self.alpha,gradient)))
            #初始化元梯度
            meta_gradient = np.zeros(self.theta.shape)
            for i in range(self.num_tasks):
                #抽取k个数据点，并为元训练准备训练集
                XTest, YTest = sample_points(10)

                #预测y的值
                a = np.matmul(XTest, self.theta_[i])
                YPred = self.sigmoid(a)
```

```
                #计算元梯度
                meta_gradient += np.matmul(XTest.T, (YPred - YTest)) /
self.num_samples

            #使用元梯度更新随机初始化的模型参数theta
            self.theta = self.theta-self.beta*meta_gradient/self.num_tasks
            #使用元梯度更新随机初始化的模型参数alpha
            self.alpha = self.alpha-self.beta*meta_gradient/self.num_tasks
            if e%1000==0:
                print "Epoch {}: Loss {}\n".format(e,loss)
                print 'Updated Model Parameter Theta\n'
                print 'Sampling Next Batch of Tasks \n'
                print '---------------------------------\n'
```

创建 `MetaSGD` 类的实例：

```
model = MetaSGD()
```

开始训练模型：

```
model.train()
```

如你所见，损失随着训练轮数增加而减小：

```
Epoch 0: Loss 2.22523195333

Updated Model Parameter Theta

Sampling Next Batch of Tasks

---------------------------------

Epoch 1000: Loss 1.951785305709

Updated Model Parameter Theta

Sampling Next Batch of Tasks

---------------------------------

Epoch 2000: Loss 1.47382270343

Updated Model Parameter Theta

Sampling Next Batch of Tasks

---------------------------------

Epoch 3000: Loss 1.07296354822

Updated Model Parameter Theta

Sampling Next Batch of Tasks

---------------------------------
```

### 7.1.2　强化学习中的 Meta-SGD

这一节我们将了解如何在强化学习场景中使用 Meta-SGD。Meta-SGD 与任何可以通过梯度下降训练的强化学习算法兼容。

(1) 假设有一个由 $\theta$ 影响的模型 $f$ 以及任务的分布 $p(T)$。首先，随机初始化模型参数 $\theta$，并将 $\alpha$ 初始化为与 $\theta$ 相同的形状。

(2) 现在，从任务的分布中抽取一批任务 $T_i$，即 $T_i \sim p(T)$。假设抽取了 3 个任务，$T = \{T_1, T_2, T_3\}$。

(3) **内循环**：对于任务 $T$ 中的每个任务 $T_i$，抽取 $D_i^{\text{train}}$ 轨迹，计算损失，并利用梯度下降最小化损失，得到最优参数 $\theta_i' = \theta - \boldsymbol{\alpha} \circ \nabla_\theta L_{T_i}(f_\theta)$。对于每个任务，抽取轨迹，最小化损失，并得到最优参数 $\theta_i'$。当抽取 3 个任务时，我们会有 3 个最优参数 $\{\theta_1', \theta_2', \theta_3'\}$。接下来我们将抽取另一个轨迹集合 $D_i^{\text{test}}$ 用于元更新。

(4) **外循环**：现在在 $D_i^{\text{test}}$ 轨迹中执行元优化。通过计算相对于上一步最优参数 $\theta_i'$ 的梯度以减少损失。我们不仅更新 $\theta$，还更新随机初始化的参数 $\boldsymbol{\alpha}$：

$$\theta = \theta - \beta \nabla_\theta \sum_{T_i \sim p(T)} L_{T_i}(f_{\theta_i'})$$

$$\boldsymbol{\alpha} = \boldsymbol{\alpha} - \beta \nabla_{\boldsymbol{\alpha}} \sum_{T_i \sim p(T)} L_{T_i}(f_{\theta_i'})$$

(5) 对步骤(2)~(4)进行 $n$ 次迭代。

## 7.2　Reptile

Reptile 算法是 OpenAI 对 MAML 算法的改进，它简单且易于实现。我们知道，在 MAML 中可以计算二阶导数，也就是梯度的梯度，但在计算上，这不是有效的任务。因此，OpenAI 提出了 MAML 的改进算法——Reptile。该算法非常简单。对 $n$ 个任务进行抽样，对每个抽样任务进行迭代次数更少的**随机梯度下降**（SGD），然后按照所有任务的共同方向更新模型参数。由于对每个任务执行迭代次数更少的 SGD，这意味着我们在计算损失的二阶导数，但与 MAML 不同，它在计算上是有效的，因为没有直接计算二阶导数，也没有展开计算图，所以它更容易实现。

假设我们从任务的分布中抽取了两个任务 $T_1$ 和 $T_2$，并随机初始化模型参数。首先，对任务 $T_1$ 执行 $n$ 次 SGD，得到最优参数 $\theta_1'$；然后对任务 $T_2$ 执行 $n$ 次 SGD，得到最优参数 $\theta_2'$。因此，我们有两组最优参数：$\theta' = \{\theta_1', \theta_2'\}$。现在需要将参数 $\theta$ 移动到更接近这两个最优参数的方向，如图 7-1 所示。

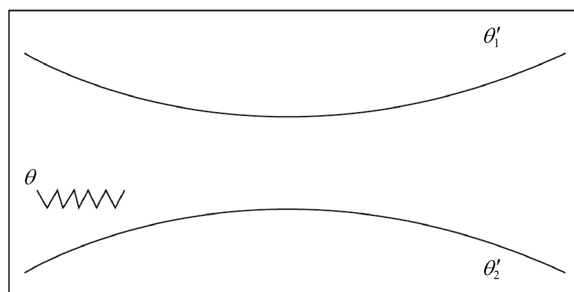



图 7-1

但是如何才能将随机初始化的模型参数$\theta$移动到更接近最优参数$\theta'$的方向呢？首先，需要找到随机初始化的模型参数$\theta$与最优参数集$\theta'$之间的距离。用欧氏距离$D$来表示这个距离。一旦找到$\theta$和$\theta'$之间的距离，需要最小化它们：

$$\mathrm{Min}_{\theta}\mathbb{E}[\frac{1}{2}D(\theta,\theta')^2]$$

在最小化$\theta$和$\theta'$之间距离的过程中，随机初始化的模型参数$\theta$，将其向更靠近最优参数$\theta'$的方向移动，但如何最小化这个距离呢？通过计算距离的梯度$\nabla_{\theta}\mathbb{E}[\frac{1}{2}D(\theta,\theta')^2]$来最小化距离，可以写成如下：

$$\theta=\theta-\epsilon\nabla_{\theta}\mathbb{E}[\frac{1}{2}D(\theta,\theta')^2]$$

计算梯度之后，最终的更新方程如下：

$$\theta=\theta+\epsilon(\theta'-\theta)$$

通过使用上述方程更新模型参数$\theta$，最小化了初始参数$\theta$和最优参数值$\theta'$之间的距离。因此，通过执行$n$次 SGD 迭代，找到每个任务的最优参数。一旦得到了这个最优参数集，我们就使用前面的方程更新模型参数$\theta$。



## 7.2.1 Reptile 算法

Reptile 是一种简单而有效的算法。它可以实现串行（serial）版本和批量（batch）版本。在串行版本中，我们只对任务的分布中的一个任务进行抽样，而在批量版本中，我们对一批任务进行抽样，并试图找到最优参数。下面来看 Reptile 的串行版本是如何运作的，其步骤如下。

(1) 假设有任务的分布$p(T)$。首先，随机初始化参数$\theta$。

(2) 现在，从任务中抽取一个任务 $T$ ，即 $T \sim p(T)$ 。

(3) **内循环**：对于任务 $T$ ，抽取 $k$ 个数据点并准备数据集：

$$D = \{(x_1, y_1), (x_2, y_2), \cdots, (x_k, y_k)\}$$

该数据集包含 $x$ 个特征和 $y$ 个标签。现在，通过对随机梯度下降执行 $n$ 次迭代来最小化数据集中的损失。在任务 $T$ 上进行 $n$ 次 SGD 迭代后，得到最优参数 $\theta'$ 。

(4) 使用在上一步 $\theta = \theta + \epsilon(\theta' - \theta)$ 中得到的最优参数 $\theta'$ ，朝着更靠近最优参数 $\theta'$ 的方向更新随机初始化的参数 $\theta$ 。

(5) 对步骤(2)~(4)进行 $n$ 次迭代。

### 7.2.2 使用 Reptile 进行正弦曲线回归

在上一节，我们了解了 Reptile 的运作原理。现在，通过从头开始编写代码，我们将更好地理解 Reptile。假设有一组任务，每个任务的目标是根据给定的输入，输出回归后的正弦曲线。该如何理解这句话呢?

设 $y = 振幅 \times \sin(x + 相位)$ 。Reptile 算法的目标是学习对给定 $x$ 的情况下对 $y$ 值进行回归，振幅值（amplitude value）在 0.1~5.0 范围内随机选取，相位值（phase value）在 $0 \sim \pi$ 范围内随机选取。因此，对于每个任务，我们只抽取 10 个数据点并训练网络，也就是说，对于每个任务，只抽样 10 对`(x, y)`。让我们来看看详细代码。

你也可以在 GitHub 网站搜索 Hands-On-Meta-Learning-With-Python，在带有注释的 Jupyter Notebook 中查看相应代码。

首先，导入所有需要的库：

```
import tensorflow as tf
import numpy as np
```

#### 1. 生成数据点

现在定义一个名为 `sample_points` 的函数来生成`(x, y)`对。它以参数 `k` 为输入，表示要抽取的`(x, y)`对的个数：

```
def sample_points(k):
    num_points = 100
    #振幅
    amplitude = np.random.uniform(low=0.1, high=5.0)
    #相位
    phase = np.random.uniform(low=0, high=np.pi)

    x = np.linspace(-5, 5, num_points)
```

```
    #y = axsin(x+b)
    y = amplitude * np.sin(x + phase)
    #抽取k个数据点
    sample = np.random.choice(np.arange(num_points), size=k)
    return (x[sample], y[sample])
```

### 2. 双层神经网络

与 MAML 一样，Reptile 与任何可以通过梯度下降训练的算法兼容。因此，使用简单的包含 64 个隐藏单元的双层神经网络。

首先，重置 TensorFlow 图：

```
tf.reset_default_graph()
```

初始化网络参数：

```
num_hidden = 64
num_classes = 1
num_feature = 1
```

接下来，为输入和输出定义占位符：

```
X = tf.placeholder(tf.float32, shape=[None, num_feature])
Y = tf.placeholder(tf.float32, shape=[None, num_classes])
```

随机初始化模型参数：

```
w1 = tf.Variable(tf.random_uniform([num_feature, num_hidden]))
b1 = tf.Variable(tf.random_uniform([num_hidden]))

w2 = tf.Variable(tf.random_uniform([num_hidden, num_classes]))
b2 = tf.Variable(tf.random_uniform([num_classes]))
```

然后，执行前馈操作来预测输出 `Yhat`：

```
#1层
z1 = tf.matmul(X, w1) + b1
a1 = tf.nn.tanh(z1)

#输出层
z2 = tf.matmul(a1, w2) + b2
Yhat = tf.nn.tanh(z2)
```

使用均方误差作为损失函数：

```
loss_function = tf.reduce_mean(tf.square(Yhat - Y))
```

使用 Adam 优化器使损失最小：

```
optimizer = tf.train.AdamOptimizer(1e-2).minimize(loss_function)
```

初始化 TensorFlow 变量：

```
init = tf.global_variables_initializer()
```

### 3. Reptile

下面我们来看看如何用 Reptile 找到神经网络的最优参数。

首先，初始化所有需要的变量：

```
#轮数，即训练迭代次数
num_epochs = 100

#样本数
num_samples = 50

#任务数
num_tasks = 2

#执行优化的次数
num_iterations = 10

#小批量大小
mini_batch = 10
```

然后，启动 TensorFlow 会话：

```
with tf.Session() as sess:
    sess.run(init)
```

对于每轮训练：

```
for e in range(num_epochs):
    #对于一批任务中的每个任务
    for task in range(num_tasks):
```

初始化模型参数：

```
old_w1, old_b1, old_w2, old_b2 = sess.run([w1, b1, w2, b2,])
```

抽样 x 和 y：

```
x_sample, y_sample = sample_points(num_samples)
```

对任务执行 *k* 次优化：

```
            for k in range(num_iterations):

                #获取小批量的x和y
                for i in range(0, num_samples, mini_batch):

                #抽取小批量样本
                x_minibatch = x_sample[i:i+mini_batch]
                y_minibatch = y_sample[i:i+mini_batch]
```

```
                train = sess.run(optimizer, feed_dict={X:
x_minibatch.reshape(mini_batch,1),
                                                        Y:
y_minibatch.reshape(mini_batch,1)})
```

在几轮优化后，得到更新后的模型参数：

```
new_w1, new_b1, new_w2, new_b2 = sess.run([w1, b1, w2, b2])
```

执行元更新：

```
epsilon = 0.1

updated_w1 = old_w1 + epsilon * (new_w1 - old_w1)
updated_b1 = old_b1 + epsilon * (new_b1 - old_b1)

updated_w2 = old_w2 + epsilon * (new_w2 - old_w2)
updated_b2 = old_b2 + epsilon * (new_b2 - old_b2)
```

使用新参数更新模型参数：

```
w1.load(updated_w1, sess)
b1.load(updated_b1, sess)

w2.load(updated_w2, sess)
b2.load(updated_b2, sess)
```

每 10 轮打印一次损失：

```
        if e%10 == 0:
            loss = sess.run(loss_function, feed_dict={X:
x_sample.reshape(num_samples,1), Y: y_sample.reshape(num_samples,1)})

            print "Epoch {}: Loss {}\n".format(e,loss)
            print 'Updated Model Parameter Theta\n'
            print 'Sampling Next Batch of Tasks \n'
            print '---------------------------------\n'
```

完整的代码如下：

```
#启动 TensorFlow 会话
with tf.Session() as sess:
    sess.run(init)
    for e in range(num_epochs):
        #对于一批任务中的每个任务
        for task in range(num_tasks):
            #初始化模型参数
            old_w1, old_b1, old_w2, old_b2 = sess.run([w1, b1, w2, b2,])

            #抽样 x 和 y
            x_sample, y_sample = sample_points(num_samples)

            #对任务执行 k 次优化
```

7

```
            for k in range(num_iterations):

                #获取小批量的 x 和 y
                for i in range(0, num_samples, mini_batch):

                    #抽取小批量样本
                    x_minibatch = x_sample[i:i+mini_batch]
                    y_minibatch = y_sample[i:i+mini_batch]

                    train = sess.run(optimizer, feed_dict={X:
x_minibatch.reshape(mini_batch,1),
                                                           Y:
y_minibatch.reshape(mini_batch,1)})

            #在几轮优化后，得到更新后的模型参数
            new_w1, new_b1, new_w2, new_b2 = sess.run([w1, b1, w2, b2])

            #执行元更新

            #即 theta = theta + epsilon × (theta_star - theta)

            epsilon = 0.1

            updated_w1 = old_w1 + epsilon * (new_w1 - old_w1)
            updated_b1 = old_b1 + epsilon * (new_b1 - old_b1)

            updated_w2 = old_w2 + epsilon * (new_w2 - old_w2)
            updated_b2 = old_b2 + epsilon * (new_b2 - old_b2)

            #使用新参数更新模型参数
            w1.load(updated_w1, sess)
            b1.load(updated_b1, sess)

            w2.load(updated_w2, sess)
            b2.load(updated_b2, sess)
            if e%10 == 0:
                loss = sess.run(loss_function, feed_dict={X:
    x_sample.reshape(num_samples,1), Y: y_sample.reshape(num_samples,1)})

                print "Epoch {}: Loss {}\n".format(e,loss)
                print 'Updated Model Parameter Theta\n'
                print 'Sampling Next Batch of Tasks \n'
                print '---------------------------------\n'
```

输出如下：

```
Epoch 0: Loss 13.0675544739

Updated Model Parameter Theta

Sampling Next Batch of Tasks

---------------------------------
```

```
Epoch 10: Loss 7.3604927063

Updated Model Parameter Theta

Sampling Next Batch of Tasks

---------------------------------

Epoch 20: Loss 4.35141277313

Updated Model Parameter Theta

Sampling Next Batch of Tasks

---------------------------------
```

## 7.3 小结

在本章，我们学习了 Meta-SGD 和 Reptile 算法；看到了如何从头构建 Meta-SGD，它在监督学习和强化学习中的应用，及其如何在学习模型参数的同时，学习学习率和更新方向；了解了 Reptile 与 MAML 的不同之处，以及前者是如何改进后者的；掌握了如何在正弦曲线回归任务中使用 Reptile。

下一章将介绍如何使用梯度一致作为元学习中的优化目标。

## 7.4 思考题

(1) Meta-SGD 与 MAML 有何不同？
(2) Meta-SGD 是如何找到最优参数的？
(3) Meta-SGD 中学习率的更新方程是什么？
(4) Reptile 算法是如何运作的？
(5) Reptile 算法的更新方程是什么？



## 7.5 延伸阅读

- Meta-SGD：参见 Zhenguo Li、Fengwei Zhou、Fei Chen 等人的文章 *Meta-SGD: Learning to Learn Quickly for Few-Shot Learning*。
- Reptile：参见 Alex Nichol、Joshua Achiam 和 John Schulman 的文章 *On First-Order Meta-Learning Algorithms*。

第 8 章

# 梯度一致作为优化目标

在上一章，我们学习了 Meta-SGD 和 Reptile 算法；了解了如何使用 Meta-SGD 来寻找最优参数、最优学习率和梯度更新方向；还掌握了 Reptile 算法是如何运作的，以及它在哪些方面比 MAML 更有效。在本章，我们将学习如何将梯度一致作为元学习的优化目标。正如你在 MAML 中看到的，我们对任务间的梯度进行平均，并更新模型参数。在梯度一致算法中，我们将使用梯度的加权平均来更新模型参数，并学习如何向梯度添加权重来找到更好的模型参数。我们将详细探讨梯度一致算法是如何工作的，该算法可以用 MAML 和 Reptile 算法来实现；还将阐释如何在 MAML 中从零开始实现梯度协议。

本章内容包括：

- 梯度一致；
- 权重计算；
- 梯度一致算法；
- 使用 MAML 构建梯度一致算法。

## 8.1 梯度一致，一种优化方法

梯度一致算法是最近才被引入的算法，很有趣，它用于增强元学习算法。在 MAML 和 Reptile 中，我们试图找到更好的、可在多个相关任务间泛化的模型参数，这样就可以用更少的数据点快速学习。回忆一下前几章所学，首先随机初始化模型参数，然后从任务的分布 $p(T)$ 中随机抽样一批任务 $T_i$。对于每个抽取的任务 $T_i$，通过计算梯度最小化损失，得到更新后的参数 $\theta_i'$，形成内循环：

$$\theta_i' = \theta - \alpha \nabla_\theta L_{T_i}(f_\theta)$$

计算完所有抽样任务的最优参数后，执行元优化，即，通过在一个新的任务集合中计算损失来执行元优化。通过计算相对于最优参数 $\theta_i'$（在内循环中获得）的梯度来减少损失，并更新初始模型参数 $\theta$：

$$\theta = \theta - \beta \nabla_{\theta} \sum_{T_i \sim p(T)} L_{T_i}(f_{\theta_i'})$$

前一个方程中到底发生了什么？如果你仔细研究这个方程，就会注意到我们只是对任务间的梯度进行平均，并更新模型参数 $\theta$，这意味着所有任务在更新模型参数方面贡献均等。

但这有什么问题呢？假设我们已抽取了 4 个任务，其中 3 个任务在一个方向上有梯度更新，但是另一个任务在一个完全不同于其他任务的方向上有梯度更新。这种不一致会对更新模型初始参数产生严重影响，这是因为所有任务梯度对更新模型参数贡献均等。如图 8-1 所示，从 $T_1$ 到 $T_3$ 的所有任务梯度都在同一个方向，而任务 $T_4$ 的梯度与其他任务的梯度方向完全不同。

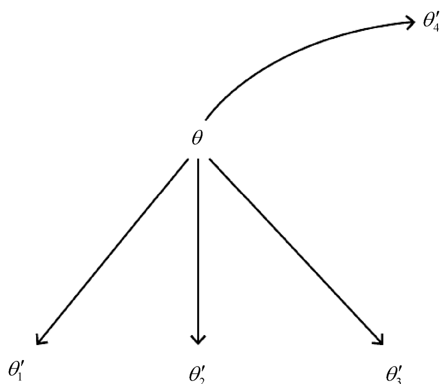



图　8-1

那么，现在应该做什么呢？如何判别任务的梯度一致呢？如果把权重和梯度联系起来，能够体现不同梯度一致的重要性吗？因此，我们通过添加与每个梯度相乘的权重，重写了外部梯度更新方程，如下：

$$\theta = \theta - \beta \sum_i \boldsymbol{w}_i \nabla L_{T_i}(f_{\theta_i})$$

如何计算这些权重呢？这些权重与任务梯度的内积以及抽样批任务中所有任务梯度的平均值成正比。而这意味着什么呢？

这意味着，**如果一个任务的梯度与抽样批任务中所有任务的平均梯度方向相同，那么可以增加它的权重，这样它对更新模型参数的贡献就会变大。类似地，如果一个任务的梯度与抽样批任务中所有任务的平均梯度有很大的不同，那么可以减少它的权重，这样它对更新模型参数的贡献就会变小。**下一节将介绍如何精确地计算这些权重。

梯度一致算法既可以应用于 MAML，又可以应用于 Reptile 算法。因此，Reptile 更新方程如下：

$$\theta = \theta + \alpha \sum_i \boldsymbol{w}_i (\theta_i' - \theta)$$

### 8.1.1　权重计算

我们已经看到，通过将权重与梯度联系起来，可以体现任务的梯度一致与梯度不一致。

我们知道，这些权重与任务梯度的内积以及抽样批任务中所有任务梯度的平均值成正比。该如何计算这些权重呢?

权重计算如下：

$$\boldsymbol{w}_i = \frac{\sum_{j \in T}(\boldsymbol{g}_i^T \boldsymbol{g}_j)}{\sum_{k \in T} | \sum_{j \in T}(\boldsymbol{g}_k^T \boldsymbol{g}_j)}$$

假设抽样了一批任务，然后，对这批任务中的每个任务采样 $k$ 个数据点，计算损失，更新梯度，找到每个任务的最优参数 $\theta_i'$ 。同时，还在 $\boldsymbol{g}_i$ 中存储了每个任务的梯度更新向量，计算方程为 $\boldsymbol{g}_i = \theta - \theta_i'$ 。

因此，第 $i$ 个任务的权重是 $\boldsymbol{g}_i$ 和 $\boldsymbol{g}_j$ 的内积和除以归一化因子。归一化因子与 $\boldsymbol{g}_i$ 和 $\boldsymbol{g}_{\text{average}}$ 的内积成正比。

让我们通过下面的代码更好地理解如何精确地计算这些权重：

```
for i in range(num_tasks):
    g = theta - theta_[i]

#计算归一化因子
normalization_factor = 0

for i in range(num_tasks):
     for j in range(num_tasks):
         normalization_factor += np.abs(np.dot(g[i].T, g[j]))

#计算权重
w = np.zeros(num_tasks)

for i in range(num_tasks):
     for j in range(num_tasks):
         w[i] += np.dot(g[i].T, g[j])

     w[i] = w[i] / normalization_factor
```

### 8.1.2　算法

现在让我们逐步了解梯度一致的原理。

(1) 假设有一个由 $\theta$ 影响的模型 $f$ 以及任务的分布 $p(T)$ 。首先，随机初始化参数 $\theta$ 。

(2) 从任务的分布中抽取一批任务 $T_i$ ，即 $T_i \sim p(T)$ 。假设我们抽取了两个任务 $T = \{T_1, T_2\}$ 。

(3) **内循环**：对于任务 $T$ 中的每个任务 $T_i$，抽取 $k$ 个数据点并准备训练集与测试集：

$$D_i^{\text{train}} = \{(x_1, y_1), (x_2, y_2), \cdots, (x_k, y_k)\}$$
$$D_i^{\text{test}} = \{(x_1, y_1), (x_2, y_2), \cdots, (x_k, y_k)\}$$

在 $D_i^{\text{train}}$ 上计算损失，并利用梯度下降最小化损失，得到最优参数 $\theta_i'$：

$$\theta_i' = \theta - \alpha \nabla_\theta L_{T_i}(f_\theta)$$

同时，将梯度更新向量存储为 $\boldsymbol{g}_i = \theta - \theta_i'$。

对于每个任务，抽取 $k$ 个数据点，最小化训练集 $D_i^{\text{train}}$ 上的损失，并得到最优参数 $\theta_i'$。当抽取两个任务时，我们将有两个最优参数 $\theta' = \{\theta_1', \theta_2'\}$，以及对应于这两个参数的梯度更新向量 $\boldsymbol{g} = \{(\theta - \theta_1'), (\theta - \theta_2')\}$。

(4) **外循环**：现在，在执行元优化之前，计算权重：

$$\boldsymbol{w}_i = \frac{\sum_{j \in T} (\boldsymbol{g}_i^T \boldsymbol{g}_j)}{\sum_{k \in T} | \sum_{j \in T} (\boldsymbol{g}_k^T \boldsymbol{g}_j)}$$

计算权重后，现在将权重与梯度联系起来以执行元优化。通过计算相对于前一步得到的参数的梯度，最小化 $D_i^{\text{test}}$ 的损失，并将梯度与权重相乘。

如果元学习算法是 MAML，则更新后方程为

$$\theta = \theta - \beta \sum_i \boldsymbol{w}_i \nabla L_{T_i}(f_{\theta_i})$$

如果元学习算法是 Reptile，则更新方程为

$$\theta = \theta + \alpha \sum_i \boldsymbol{w}_i (\theta' - \theta)$$

(5) 对步骤(2)~(5)进行 $n$ 次迭代。

## 8.2　使用 MAML 构建梯度一致

在上一节，我们了解了梯度一致算法的工作原理，看到了梯度一致如何给梯度添加权重以体现不同梯度的重要性。在这一节，我们将从头开始，使用 NumPy 编码，来学习如何将梯度一致算法与 MAML 结合。为了更好地理解，我们将考虑一个简单的二分类任务。我们将随机生成输入数据，用简单的单层神经网络训练它，并试图找到最优参数 $\theta$。

下面我们将逐步学习具体的做法。

你也可以在 GitHub 网站搜索 Hands-On-Meta-Learning-With-Python，在带有注释的 Jupyter Notebook 中查看相应代码。

首先，导入所有需要的库：

```
import numpy as np
```

### 8.2.1　生成数据点

下面定义一个名为 sample_points 的函数来生成输入(x,y)对。它以参数 k 为输入，表示抽取的(x,y)对的个数：

```
def sample_points(k):
    x = np.random.rand(k,50)
    y = np.random.choice([0, 1], size=k, p=[.5, .5]).reshape([-1,1])
    return x,y
```

### 8.2.2　单层神经网络

简单起见，使用单层神经网络来预测输出：

```
a = np.matmul(X, theta)
YHat = sigmoid(a)
```

因此，我们使用 MAML 的梯度一致算法来寻找可在任务间泛化的最优参数值 theta。因此，对于新任务，可以通过更少的梯度步骤在更短的时间内从少量数据点中学习。

### 8.2.3　MAML 中的梯度一致

下面定义一个名为 GradientAgreement_MAML 的类，在这个类中我们将实现梯度一致 MAML 算法。在__init__方法中，我们会初始化所有必要的变量。然后定义 sigmoid 激活函数与 train 函数。

让我们一步一步来：

```
class GradientAgreement_MAML(object):
```

定义__init__方法，并初始化所有必要的变量。

```
    def __init__(self):
        #初始化任务的数量，即每批任务中我们需要的任务数量
        self.num_tasks = 2
        #样本的数量，即每个任务中我们需要的数据点数量（k）
        self.num_samples = 10
        #轮数，即训练迭代次数
        self.epochs = 100
        #内循环（内部梯度更新）的超参数
```

```
        self.alpha = 0.0001
        #外循环（外部梯度更新），即元更新的超参数
        self.beta = 0.0001
        #随机初始化模型参数theta
        self.theta =
np.random.normal(size=self.pol_ord).reshape(self.pol_ord, 1)
```

定义 sigmoid 激活函数，用于将 x 转化为多项式形式：

```
def sigmoid(self,a):
    return 1.0 / (1 + np.exp(-a))
```

现在定义 train 函数用于训练：

```
def train(self):
```

对于每轮训练：

```
for e in range(self.epochs):
    self.theta_ = []
    #用于存储梯度更新
    self.g = []
```

对于每批任务中的任务 i：

```
for i in range(self.num_tasks):
```

抽取 $k$ 个数据点，并准备训练集 $D_i^{\text{train}}$：

```
XTrain, YTrain = sample_points(self.num_samples)
```

然后，预测 YHat 的值：

```
a = np.matmul(XTrain, self.theta)

YHat = self.sigmoid(a)
```

使用梯度下降 $\theta_i' = \theta - \alpha \nabla_\theta L_{T_i}(f_\theta)$ 计算损失并最小化损失：

```
#由于在执行分类，我们使用交叉熵损失作为损失函数
loss = ((np.matmul(-YTrain.T, np.log(YHat)) - np.matmul((1 -YTrain.T), np.log(1 -
YHat)))/self.num_samples)[0][0]
#通过计算梯度来最小化损失
gradient = np.matmul(XTrain.T, (YHat - YTrain)) / self.num_samples

#更新梯度并寻找每个任务的最优参数theta'
self.theta_.append(self.theta - self.alpha*gradient)
```

在 g 中存储梯度更新 $\boldsymbol{g}_i = \theta - \theta_i'$：

```
self.g.append(self.theta-self.theta_[i])
```

8

现在，计算权重，$\boldsymbol{w}_i = \frac{\sum_{j\in T}(\boldsymbol{g}_i^T \boldsymbol{g}_j)}{\sum_{k\in T}|\sum_{j\in T}(\boldsymbol{g}_k^T \boldsymbol{g}_j)|}$：

```
            normalization_factor = 0
            for i in range(self.num_tasks):
                for j in range(self.num_tasks):
                    normalization_factor += np.abs(np.dot(self.g[i].T,
self.g[j]))
            w = np.zeros(self.num_tasks)
            for i in range(self.num_tasks):

                for j in range(self.num_tasks):
                    w[i] += np.dot(self.g[i].T, self.g[j])

                w[i] = w[i] / normalization_factor
```

初始化加权元梯度：

```
weighted_gradient = np.zeros(self.theta.shape)
```

为每个任务抽取 $k$ 个数据点，并准备测试集 $D_i^{\text{test}}$：

```
for i in range(self.num_tasks):

    #抽取 k 个数据点，并为元训练准备测试集
    XTest, YTest = sample_points(10)
```

预测 y 的值：

```
a = np.matmul(XTest, self.theta_[i])
YPred = self.sigmoid(a)
```

计算元梯度：

```
                meta_gradient = np.matmul(XTest.T, (YPred - YTest)) /
self.num_samples
```

将权重乘以元梯度，并更新 $\theta$ 的值：

$$\theta = \theta - \beta \sum_i \boldsymbol{w}_i \nabla L_{T_i}(f_{\theta_i})$$

```
            weighted_gradient += np.sum(w[i]*meta_gradient)

        self.theta = self.theta-
self.beta*weighted_gradient/self.num_tasks
```

每 10 轮打印一次损失：

```
if e%10==0:
    print "Epoch {}: Loss {}\n".format(e,loss)
    print 'Updated Model Parameter Theta\n'
```

```
    print 'Sampling Next Batch of Tasks \n'
    print '---------------------------------\n'
```

完整的 GradientAgreement_MAML 类如下：

```
class GradientAgreement_MAML(object):
    def __init__(self):
        #初始化任务的数量，即每批任务中我们需要的任务数量
        self.num_tasks = 2
        #样本的数量，即每个任务中我们需要的数据点数量（k）
        self.num_samples = 10

        #轮数，即训练迭代次数
        self.epochs = 100
        #内循环（内部梯度更新）的超参数
        self.alpha = 0.0001
        #外循环（外部梯度更新），即元更新的超参数
        self.beta = 0.0001
        #随机初始化模型参数 theta
        self.theta = np.random.normal(size=50).reshape(50, 1)
    #定义 sigmoid 激活函数
    def sigmoid(self,a):
        return 1.0 / (1 + np.exp(-a))
    #下面定义 train 函数用于训练
    def train(self):
        #对于每轮训练
        for e in range(self.epochs):
            self.theta_ = []
            #用于存储梯度更新
            self.g = []
            #对于每批任务中的任务 i
            for i in range(self.num_tasks):
                #抽取 k 个数据点，并准备训练集
                XTrain, YTrain = sample_points(self.num_samples)
                a = np.matmul(XTrain, self.theta)

                YHat = self.sigmoid(a)

                #由于在执行分类，我们使用交叉熵损失作为损失函数
                loss = ((np.matmul(-YTrain.T, np.log(YHat)) - np.matmul((1
-YTrain.T), np.log(1 - YHat)))/self.num_samples)[0][0]
                #通过计算梯度来最小化损失
                gradient = np.matmul(XTrain.T, (YHat - YTrain)) /
self.num_samples

                #更新梯度并寻找每个任务的最优参数 theta'
                self.theta_.append(self.theta - self.alpha*gradient)
                #计算梯度更新
                self.g.append(self.theta-self.theta_[i])
             #现在计算权重
             #我们知道权重是 g_i 和 g_j 的点积和除以归一化因子
             normalization_factor = 0
             for i in range(self.num_tasks):
                 for j in range(self.num_tasks):
```

```
                    normalization_factor += np.abs(np.dot(self.g[i].T,
self.g[j]))
            w = np.zeros(self.num_tasks)
            for i in range(self.num_tasks):

                for j in range(self.num_tasks):
                    w[i] += np.dot(self.g[i].T, self.g[j])

                w[i] = w[i] / normalization_factor
            #初始化元梯度
            weighted_gradient = np.zeros(self.theta.shape)
            for i in range(self.num_tasks):
                #抽取k个数据点，并为元训练准备测试集
                XTest, YTest = sample_points(10)
                #预测y的值
                a = np.matmul(XTest, self.theta_[i])
                YPred = self.sigmoid(a)
                #计算元梯度
                meta_gradient = np.matmul(XTest.T, (YPred - YTest)) /
self.num_samples
                weighted_gradient += np.sum(w[i]*meta_gradient)

            #使用元梯度更新随机初始化的模型参数theta
            self.theta = self.theta-
self.beta*weighted_gradient/self.num_tasks
            if e%10==0:
                print "Epoch {}: Loss {}\n".format(e,loss)
                print 'Updated Model Parameter Theta\n'
                print 'Sampling Next Batch of Tasks \n'
                print '---------------------------------\n'
```

创建 GradientAgreement_MAML 类的实例：

```
model = GradientAgreement_MAML()
```

然后，训练模型：

```
model.train()
```

可以看到损失是如何随着训练轮数的增加而降低的：

```
Epoch 0: Loss 5.9436043239

Updated Model Parameter Theta

Sampling Next Batch of Tasks

---------------------------------

Epoch 10: Loss 3.905350606769

Updated Model Parameter Theta

Sampling Next Batch of Tasks
```

```
--------------------------------
Epoch 20: Loss 2.0736155578
Updated Model Parameter Theta
Sampling Next Batch of Tasks
--------------------------------
Epoch 30: Loss 1.48478751777
Updated Model Parameter Theta
Sampling Next Batch of Tasks
--------------------------------
```

## 8.3　小结

在本章，我们学习了梯度一致算法；了解了梯度一致算法如何使用加权梯度来找到更好的初始模型参数$\theta$；分析了这些权重是如何与任务梯度的内积以及抽取的批任务中所有任务梯度的平均值成正比的；之后探讨了如何将梯度一致算法与 MAML 算法和 Reptile 算法相结合；最后研究了如何使用梯度一致算法在分类任务中找到最优参数$\theta$。

在下一章，我们将了解一些元学习的新进展，如任务无关元学习（task agnostic meta learning）、概念空间元学习（learning to learn in the concept space）和元模仿学习（meta imitation learning）。

## 8.4　思考题

(1) 什么是梯度一致与梯度不一致？
(2) 梯度一致中 MAML 的更新方程是什么？
(3) 梯度一致的权重是什么？
(4) 如何计算权重？
(5) 什么是归一化因子？
(6) 何时增减权重？



## 8.5　延伸阅读

- 梯度一致算法的论文： Amir Erfan Eshratifar、David Eigen 和 Massoud Pedram 的文章 *Gradient Agreement as an Optimization Objective for Meta-Learning*。

第 9 章

# 新进展与未来方向



恭喜你！终于读到了最后一章。我们已经走了很长一段路：从元学习基础开始，遇到了一些单样本学习算法，如孪生网络、原型网络、匹配网络和关系网络；然后了解了 NTM 是如何存储和检索信息的；之后认识了一些有趣的元学习算法，如 MAML、Reptile 和 Meta-SGD，以及它们是如何找到最优初始参数的。这一章，我们将了解元学习的一些新进展；学习如何使用任务无关元学习（task agnostic meta learning，TAML）来减少元学习中的任务偏差（bias），以及如何在模仿学习系统（imitation learning system）中使用元学习；然后分析如何使用 CACTUs 算法将 MAML 应用到无监督学习场景中；最后学习一种叫作概念空间元学习（learning to learn in the concept space）的深度元学习算法。

本章内容包括：

- TAML；
- 元模仿学习；
- CACTUs；
- 概念空间元学习。

## 9.1 TAML

我们知道，在元学习中，通过相关任务的分布来训练模型，这样它就可以很容易地适应新任务，且只需要几个样本。在前几章，我们已了解了 MAML 如何通过计算元梯度和执行元优化来找到模型的最优初始参数。但是，我们可能会面临一个问题：模型可能会在某些任务上有偏（biased），特别是在元训练阶段抽样的任务上。因此，模型会在这些任务上过度执行（overperform）。如果出现了这种情况，它还将阻碍我们寻找更好的更新规则。对于在某些任务上有偏的模型，我们也不能更好地在那些与元训练任务有很大差异的不可见任务上进行泛化。

为了改善这种情况，我们需要使模型在某些任务上无偏或不过度执行，也就是说，需要使模型与任务无关，来防止任务偏差并获得更好的泛化。现在，我们将看到执行 TAML 的两种算法：

- ❑ 熵最大化（entropy maximization）/熵约简（entropy reduction）
- ❑ 不平等最小化（inequality minimization）

### 9.1.1 熵最大化/熵约简

在本节，我们将学习如何通过最大化和最小化熵来预防任务偏差。我们知道熵是一种对随机性的度量。因此，通过允许模型对预测标签（predicted label）进行等概率的随机猜测来最大化熵。通过对预测标签进行随机猜测，可以预防任务偏差。

如何计算熵呢？将熵记为 $H$ 。我们基于 $N$ 个预测标签的输出概率 $y_{i,n}$ ，从 $p_{T_i}(x_i)$ 中抽取 $x_i$ 来计算熵 $T_i$ ：

$$H_{T_i}(f_\theta) = -\mathbb{E}_{x_i \sim p_{T_i}(x)} \sum_{n=1}^{N} \hat{y}_{i,n} \log(\hat{y}_{i,n})$$

上式中， $\hat{y}_i$ 代表模型预测的标签。

因此，我们先最大化熵，然后更新模型参数以最小化熵。那么，最小化熵是什么意思呢？这意味着我们不会在预测的标签上添加任何随机性，并且允许模型以高可信度预测标签。

因此，我们的目标是最大限度地降低每个任务的熵，如下所示：

$$H_{T_i}(f_\theta) - H_{T_i}(f_{\theta_i})$$

我们将熵项（entropy term）与元目标（meta objective）相结合，并试图找到最优参数 $\theta$ ，因此元目标如下：

$$\theta = \theta - \beta \nabla_\theta \{ \mathbb{E}_{T_i \sim p(T)} L_{T_i}(f_{\theta_i'}) + \lambda [-H_{T_i}(f_\theta) + H_{T_i}(f_{\theta_i})] \}$$

$\lambda$ 是这两项之间的平衡系数。

**算法**

下面让我们逐步了解熵 TAML 的原理。

(1) 假设有一个由 $\theta$ 影响的模型 $f$ 以及任务的分布 $p(T)$ 。首先，随机初始化模型参数 $\theta$ 。
(2) 从任务的分布中抽取一批任务 $T_i$ ，即 $T_i \sim p(T)$ 。假设抽取了 3 个任务， $T = \{T_1, T_2, T_3\}$ 。
(3) **内循环**：对于任务 $T$ 中的每个任务 $T_i$ ，抽取 $k$ 个数据点并准备训练集与测试集：

$$D_{\text{train}} = \{(x_1, y_1), (x_2, y_2), \cdots, (x_k, y_k)\}$$
$$D'_{\text{test}} = \{(x_1', y_1'), (x_2', y_2'), \cdots, (x_k', y_k')\}$$

然后，在 $D_{\text{train}}$ 上计算损失，并利用梯度下降最小化损失，得到最优参数：

$$\theta_i' = \theta - \alpha \nabla_\theta L_{T_i}$$

因此，对于每个任务，我们抽取 $k$ 个数据点，准备训练集，最小化损失，得到最优参数。当抽取 3 个任务时，我们会有 3 个最优参数 $\{\theta_1', \theta_2', \theta_3'\}$ 。

(4) **外循环**：执行元优化。在这里，我们尝试最小化 $D_{\text{test}_i}$ 上的损失。通过计算相对于最优参数 $\theta_i'$ 的梯度来最小化损失，并更新随机初始化的参数 $\theta$ ，同时，加上熵项。因此，最终的元目标如下：

$$\theta = \theta - \beta \nabla_\theta \{\mathbb{E}_{T_i \sim p(T)} L_{T_i}(f_{\theta_i'}) + \lambda[-H_{T_i}(f_\theta) + H_{T_i}(f_{\theta_i})]\}$$

(5) 对步骤(2)~(4)进行 $n$ 次迭代。

## 9.1.2　不平等最小化

熵方法的问题在于它只适用于分类任务。因此，我们无法将算法用于回归或强化学习任务。为了解决这个问题，我们将看到另一种算法——不平等最小化 TAML。它和熵方法一样简单。在这种方法中，我们试图使不平等最小化。经济学中有几种衡量收入分配、财富分配不平等的方法。在元学习场景中，可以使用这些经济学上的不平等度量（inequality measures）来最小化任务偏差。因此，可以通过最小化一批任务中所有抽样任务损失的不平等来最小化该模型在任务上的偏差。

### 1. 不平等度量

我们会看到一些常用的不平等度量。可以将任务 $T_i$ 中的损失定义为 $l_i$ ，抽样任务的平均损失定义为 $\bar{l}$ ，单批任务中的任务数量定义为 $M$ 。

- **基尼系数**

基尼系数是最广泛使用的不平等度量之一。它用洛伦茨曲线（Lorenz curve）来测量分配不平等。洛伦茨曲线是一种累积频率曲线（cumulative frequency curve），它将特定变量的分布与代表平等的均匀分布进行比较。基尼系数的取值范围为 0~1，其中 0 代表完全平等，1 代表完全不平等。它的值是相对的绝对平均差（relative absolute mean difference）的一半。

因此，在元学习场景中，基尼系数的计算如下：

$$G = \frac{\sum_{i=1}^{M}\sum_{j=1}^{M} |l_i - l_j|}{2n\sum\sum_{i=1}^{M} l_i}$$

- **泰尔指数**

泰尔指数是另一种常用的不平等度量，以荷兰计量经济学家亨利·泰尔（Henri Theil）的名字命名。它被称为**广义熵度量**（generalized entropy measures），是不平等度量家族的一个特例，并被定义为最大熵（maximum entropy）与观测熵（observed entropy）之差。

元学习场景中的泰尔指数计算如下：

$$T=\frac{1}{M}\sum_{i=1}^{M}\frac{l_i}{l}\ln\frac{l_i}{l}$$

- **算法的方差**

算法的方差定义如下：

$$V_L(l)=\frac{1}{M}\sum_{i=1}^{M}[\ln l_i-\ln g(l)]^2$$

上式中，$g(l)$ 表示 $l$ 的几何平均。

可以使用上述任何不平等度量来计算任务偏差。一旦使用这种不平等度量来计算任务偏差，就可以通过将不平等度量插入元目标来最小化偏差。因此，可以重写元目标如下：

$$\theta-\beta\nabla_\theta[\mathbb{E}_{T_i\sim p(T)}L_{T_i}(f_{\theta_i'})+\lambda I(L_{T_i}(f_{\theta_i'}))]$$

上式中，$I(L_{T_i}(f_{\theta_i'}))$ 代表不平等度量，$\lambda$ 是平衡系数。

### 2. 算法

下面让我们逐步了解不平等最小化 TAML 的原理。

(1) 假设有一个由 $\theta$ 影响的模型 $f$ 以及任务的分布 $p(T)$。首先，随机初始化模型参数 $\theta$。
(2) 从任务的分布中抽取一批任务 $T_i$，即 $T_i\sim p(T)$。假设抽取了 3 个任务，$T=\{T_1,T_2,T_3\}$。
(3) **内循环**：对于任务 $T$ 中的每个任务 $T_i$，抽取 $k$ 个数据点并准备训练集与测试集：

$$D_{\text{train}}=\{(x_1,y_1),(x_2,y_2),\cdots,(x_k,y_k)\}$$
$$D'_{\text{test}}=\{(x'_1,y'_1),(x'_2,y'_2),\cdots,(x'_k,y'_k)\}$$

然后，在 $D_{\text{train}}$ 上计算损失，并利用梯度下降最小化损失，得到最优参数：

$$\theta_i'=\theta-\alpha\nabla_\theta L_{T_i}$$

因此，对于每个任务，抽取 $k$ 个数据点，准备训练集，最小化损失，并得到最优参数。当抽取 3 个任务时，我们会有 3 个最优参数 $\{\theta_1',\theta_2',\theta_3'\}$。

(4) **外循环**：执行元优化。在这里，我们尝试最小化 $D_{\text{test}_i}$ 上的损失。通过计算相对于最优参数 $\theta_i'$ 的梯度来最小化损失，并更新随机初始化的参数 $\theta$，同时，加上熵项。因此，最终的元目标如下：

$$\theta = \theta - \beta \nabla_\theta [\mathbb{E}_{T_i \sim p(T)} L_{T_i}(f_{\theta_i'}) + \lambda I(L_{T_i}(f_{\theta_i'}))]$$

(5) 对步骤(2)~(4)进行 $n$ 次迭代。

## 9.2　元模仿学习

如果想让机器人变成通才，能够执行各种任务，那么它应能快速学习，但怎样才能让它快速学习呢？我们人类是如何快速学习的呢？我们能通过观察别人来轻松学习新技能。同样，如果让机器人通过观察我们的行为来学习，那么就可以很容易地让机器人高效地学习复杂的目标，而无须设计复杂的目标和奖惩函数。这种类型的学习，即从人类行为中学习，被称为模仿学习，即机器人试图模仿人类的行为。机器人不一定只从人类的行为中学习，也可以向执行任务的另一个机器人学习，或者学习人类或机器人执行任务的视频。

不过模仿学习并不像听起来那么简单。机器人将需要大量的时间和演示（demonstration）来学习目标并识别正确的策略。因此，我们将用先前的经验作为演示（训练数据）来增强机器人，这样它就不必完全从头开始学习每项技能。增加机器人先前的经验有助于其快速学习。因此，为了学习多种技能，我们需要为每种技能收集演示，也就是说，需要用特定于任务的演示数据来增强机器人。

但是，如何才能让机器人从一次演示中快速学习任务呢？可以使用元学习吗？是否可以重用演示数据并从几个相关的任务中学习，从而快速地学习新任务？为此，我们将元学习与模仿学习相结合，形成了**元模仿学习**（meta imitation learning，MIL）。使用 MIL，我们可以利用来自各种其他任务的演示数据，以便通过单个演示快速学习新任务。因此，我们仅通过单个任务演示就可以找到新任务的正确策略。

对于 MIL，可以使用任何见过的元学习算法。我们将使用 MAML 作为元学习算法，它与任何可以通过梯度下降训练的算法兼容。我们将使用策略梯度（policy gradients）作为寻找正确策略的算法。在策略梯度中，可以直接用某个参数 $\theta$ 对参数化的策略 $\pi_\theta$ 进行优化。

我们的目标是学习一种策略，该策略可以从新任务的单个演示中快速适应该任务。由此，可以消除对每个任务的大量演示数据的依赖。可这里的任务究竟是什么呢？任务会包括轨迹。轨迹（tr）由一系列来自专家策略（expert policy）的观察和操作组成，也就是演示。那什么是专家策略呢？因为我们在进行模仿学习，并从专家（人类动作）那里学习，所以把这个策略称为专家策略 $\pi^*$：

$$\text{tr} = \{o_1, a_1, \cdots, o_t, a_t\} \sim \pi_i^*$$

那应该用什么损失函数呢？损失函数代表机器人行为与专家行为之间的差异。可以用均方误差损失作为连续行为的损失函数，交叉熵作为离散行为的损失函数。假设行为连续，则可以将均方误差损失表示为

$$L_{T_i}(f_\theta) = \sum_{\text{tr}^{(j)} \sim T_i} \sum_t \left\| f_\theta(o_t)^{(j)} - a_t^{(j)} \right\|_2^2$$

假设有任务的分布 $p(T)$ 。对一批任务进行抽样，并从每个任务 $T_i$ 中抽取一些演示数据，通过最小化损失来训练网络，找到最优参数 $\theta'$ 。然后，计算元梯度进行元优化，找到最优初始参数 $\theta$ 。下一节将介绍这其中的运作原理。

## MIL 算法

MIL 的步骤如下。

(1) 假设有一个由 $\theta$ 影响的模型 $f$ 以及任务的分布 $p(T)$ 。首先，随机初始化模型参数 $\theta$ 。

(2) 从任务的分布中抽取一批任务 $T_i$ ，即 $T_i \sim p(T)$ 。

(3) **内循环**：对于被抽样任务中的每个任务，抽取演示数据：

$$\text{tr} = \{o_1, a_1, \cdots, o_t, a_t\}$$

然后，计算损失，并利用梯度下降最小化损失，得到最优参数：

$$\theta_i' = \theta - \alpha \nabla_\theta L_{T_i}(f_\theta)$$

随后，再抽取一份演示数据用于元训练：

$$\text{tr}' = \{o_1', a_1', \cdots, o_t', a_t'\}$$

(4) **外循环**：执行元优化。现在，通过元优化，使用 $\text{tr}'$ 更新初始参数：

$$\theta = \theta - \beta \nabla_\theta \sum_{T_i \sim p(T)} L_{T_i}(f_{\theta_i'})$$

(5) 对步骤(2)~(4)进行 $n$ 次迭代。

# 9.3 CACTUs

我们已看到了 MAML 如何帮助找到最优的初始模型参数，以在很多其他相关任务间泛化；了解了 MAML 在监督和强化学习环境中的应用。但如何将 MAML 应用到数据点无标记的无监督

学习环境中呢？为此，我们引入了一种新的算法——**聚类自动生成用于无监督模型无关元学习的任务**（CACTUs），或简称为**仙人掌算法**。

假设有包含未标记示例的数据集：$D=\{x_1,x_2,x_3,\cdots,x_n\}$。现在，可以用这个数据集做什么呢？如何将 MAML 应用于此数据集呢？首先，使用 MAML 训练需要哪些东西？我们需要一个任务的分布，通过对一批任务进行抽样，找到最优的模型参数来训练模型。任务应该包含一个特征及其标签，但如何才能从未标记的数据集中生成任务呢？

下一节将介绍如何使用 CACTUs 生成任务。一旦生成了任务，我们就可以很容易地将它们插入 MAML 算法，并找到最优的模型参数。

### 使用 CACTUs 生成任务

假设有包含未标记示例的数据集：$D=\{x_1,x_2,x_3,\cdots,x_n\}$。现在需要为数据集创建标签。该怎么做呢？首先，使用一些嵌入函数学习数据集中每个数据点的嵌入。嵌入函数可以是任何特征提取器。假设输入是图像，那么可以使用 CNN 作为嵌入函数来提取图像特征向量。

为每个数据点生成嵌入后，应该如何找到它们的标签呢？一种简单而朴素的方法是使用一些随机超平面将数据集 $D$ 划分为 $P$ 部分，然后可以将数据集的每个划分子集视为单独的类。

但这个方法的问题是，由于使用的是随机超平面，我们的类可能包含完全不同的嵌入，相关的嵌入也可能保存在不同的类中。因此，可以使用聚类算法，而不是使用随机超平面来划分数据集。我们使用 $k$-means 作为聚类算法来划分数据集。对 $k$-means 聚类多次迭代，得到 $k$ 个簇（cluster），或者说划分（partition）。

可以将每个簇作为单独的类来处理。因此，接下来该干什么呢？该如何生成任务呢？假设，由于聚类，我们有 5 个簇。从这 5 个簇中抽取 $n$ 个簇作为样本；然后从 $n$ 个簇中的每个簇不放回抽样 $r$ 个数据点，记为 $\{x_r\}_n$；之后抽取一个包含 $n$ 个特定于任务的独热标签的排列（permutation）$l_n$，用于为 $n$ 个抽取的簇分配标签。现在有一个数据点 $\{x_r\}_n$，和一个标签 $l_n$。

最后，可以将任务 $T$ 定义为 $T=\{(x_{n,r},l_n)\mid x_{n,r}\in\{x_r\}_n\}$。

## 9.4　概念空间元学习

现在我们来看看如何在概念空间中使用深度元学习来学习。首先，如何进行元学习呢？抽取一批相关任务，在每个任务中抽取 $k$ 个数据点，并对元学习器进行训练。可以将深度学习和元学习结合起来，而不是仅仅使用普通元学习技术进行训练。因此，当抽取一批相关任务，并在每个任务中抽取 $k$ 个数据点时，可以使用深度神经网络学习每个任务的 $k$ 个数据点的表示（representation），然后我们会对这些表示进行元学习。

概念空间元学习包括 3 个部分：

- ❑ 概念生成器（concept generator）
- ❑ 概念鉴别器（concept discriminator）
- ❑ 元学习器（meta learner）

概念生成器用于提取数据集中每个数据点的特征表示，捕捉其高层次的概念；概念鉴别器用于对概念生成器生成的概念进行识别和分类；而元学习器从概念生成器生成的概念中学习。上述 3 个部分，即概念生成器、概念鉴别器和元学习器，同时进行学习。这样，我们将元学习与深度学习相结合，改进了普通的元学习。概念生成器会随着新的输入数据而进化，因此，可以将此框架视为终生学习系统（lifelong learning system）。

但概念空间元学习究竟是如何运作的呢？请看图 9-1。如你所见，抽取一组任务，并将它们输入概念生成器，概念生成器学习概念（即嵌入），然后将这些概念提供给元学习器，元学习器学习这些概念并将损失发送回概念生成器。同时，我们还向概念生成器提供了一些外部数据集，概念生成器学习这些输入的概念并将其发送给概念鉴别器。概念鉴别器预测这些概念的标签，计算损失，并将损失发送回概念生成器。由此，增强了概念生成器泛化概念的能力。

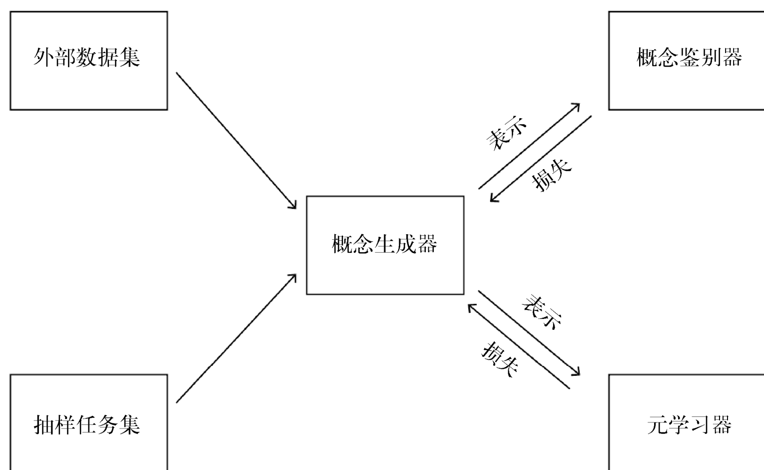



图 9-1

但是，为什么要这样做呢？这是因为我们在概念空间中进行元学习，而不是在原始数据集上进行元学习。如何学习这些概念呢？这些概念由概念生成器通过学习输入的嵌入而生成。因此，我们在各种相关任务上对概念生成器和元学习器进行训练，同时，通过向概念生成器提供外部数据集，使概念鉴别器改进概念生成器，从而概念生成器能够更好地学习概念。这种联合训练（joint training）过程使概念生成器能够学习各种概念，并在相关任务上表现得更好。提供外部数

据集只是为了提升概念生成器的性能，当我们提供一组新的输入时，概念生成器会不断地学习。因此，这是终身学习系统。

### 9.4.1 关键部分

现在让我们详细看看每个部分。

#### 1. 概念生成器

我们知道，概念生成器用于提取特征。可以使用由 $\theta_G$ 影响的深度神经网络来生成概念。例如，如果输入是图像，概念生成器可以是 CNN。

#### 2. 概念鉴别器

概念鉴别器基本上是一个分类器，用于预测概念生成器生成的概念的标签。因此，它可以是任何由 $\theta_D$ 影响的监督学习算法，如 SVM 和决策树。

#### 3. 元学习器

元学习器可以是任何由 $\theta_M$ 影响的元学习算法，比如 MAML、Meta-SGD 或 Reptile。

### 9.4.2 损失函数

下面使用两种损失函数：

- ❑ 概念鉴别损失（concept discrimination loss）
- ❑ 元学习损失

#### 1. 概念鉴别损失

我们从数据集 $\mathbb{D}$ 中抽取一些数据点 $(x, y)$，将它们输入概念生成器，概念生成器学习这些概念并将它们发送给概念鉴别器，概念鉴别器尝试为这些概念预测类。因此，概念鉴别损失代表概念鉴别器对类的预测能力，可以表示为

$$L_{(x,y)}(\theta_D, \theta_G)$$

根据任务，我们的损失函数可以是任何损失函数。例如，如果我们执行分类任务，它可能是交叉熵损失。

#### 2. 元学习损失

从任务的分布中抽取一些任务样本，通过概念生成器学习它们的概念，对这些概念进行元学习，然后计算元学习损失：

$$L_T(\theta_M, \theta_G)$$

元学习损失取决于使用的元学习器，如 MAML 或 Reptile。

最终损失函数是概念鉴别损失和元学习损失的结合：

$$损失 = L_T(\theta_M, \theta_G) + \lambda L_{(x,y)}(\theta_D, \theta_G)$$

在上式中，$\lambda$ 是元学习损失和概念鉴别损失的平衡超参数。因此，我们的目标是找到最小化这种损失的最优参数：

$$\min_{\theta_D, \theta_M, \theta_G} \mathbb{E}_{T \sim p(T),(x,y) \sim \mathbb{D}} J[L_T(\theta_M, \theta_G), L_{(x,y)}(\theta_D, \theta_G)]$$

通过计算梯度使损失最小化，并更新模型参数：

$$(\theta_D, \theta_M, \theta_G) = (\theta_D, \theta_M, \theta_G) - \beta \nabla [J(L_T(\theta_M, \theta_G), L_{(x,y)}(\theta_D, \theta_G)]$$

### 9.4.3 算法

这一节我们将逐步了解算法。

(1) 假设有任务的分布 $p(T)$ 。首先，随机初始化模型参数——概念生成器 $\theta_G$ 、元学习器 $\theta_M$ 、概念鉴别器 $\theta_D$ 。

(2) 从任务的分布中抽取一批任务样本，通过概念生成器学习它们的概念，对这些概念进行元学习，然后计算元学习损失：

$$L_T(\theta_M, \theta_G)$$

(3) 从数据集 $\mathbb{D}$ 中抽取一些数据点 $(x, y)$ ，将它们输入概念生成器，概念生成器学习这些概念并将它们发送给概念鉴别器，概念鉴别器尝试为这些概念预测类，并计算概念分类损失：

$$L_{(x,y)}(\theta_D, \theta_G)$$

(4) 将这两种损失结合起来，尝试使用 SGD 将损失最小化，并获得更新的模型参数：

$$(\theta_D, \theta_M, \theta_G) = (\theta_D, \theta_M, \theta_G) - \beta \nabla [J(L_T(\theta_M, \theta_G), L_{(x,y)}(\theta_D, \theta_G)]$$

(5) 对步骤(2)~(4)进行 $n$ 次迭代。

再次祝贺你学习了所有重要且流行的元学习算法。元学习是人工智能中一个有趣且很有前途的领域，它将使我们更接近 AGI。现在你已读完了这本书，可以开始探索元学习的各种新进展，并开始尝试各种项目了。Learn and meta learn！

## 9.5 小结

在本章，我们学习了 TAML 用于减少任务偏差，看到了两种方法——基于熵的 TAML 和基于不平等的 TAML；接下来探讨了元模仿学习，它将元学习与模仿学习相结合；然后分析了元学习如何帮助模仿学习从更少的模仿中学习，还了解了如何在使用 CACTUs 的无监督学习中应用模型无关元学习；之后探讨了一种深度元学习算法——概念空间学习；最后看到了如何通过深度学习来促进元学习。

元学习是人工智能领域中最有趣的分支之一。现在你已了解了各种元学习算法，可以开始构建元学习模型，这些模型可以在各种任务间泛化，并有助于元学习研究。

## 9.6 思考题

(1) 有哪几种不同的不平等度量？
(2) 什么是泰尔指数？
(3) 什么是模仿学习？
(4) 什么是概念生成器？
(5) 什么是元学习损失？

## 9.7 延伸阅读

- TAML：参见 Muhammad Abdullah Jamal、Guo-Jun Qi 和 Mubarak Shah 的文章 *Task-Agnostic Meta-Learning for Few-shot Learning*。
- 元模仿学习：参见 Chelsea Finn、Tianhe Yu、Tianhao Zhang 等人的文章 *One-Shot Visual Imitation Learning via Meta-Learning*。
- CACTUs：参见 Kyle Hsu、Sergey Levine 和 Chelsea Finn 的文章 *Unsupervised Learning via Meta-Learning*。
- 概念空间元学习：参见 Fengwei Zhou、Bin Wu 和 Zhenguo Li 的文章 *Deep Meta-Learning: Learning to Learn in the Concept Space*。

# 思考题答案

## 第 1 章　元学习简介

(1) 元学习能够生成通用的人工智能模型，来学习执行各种任务。我们可以用很少的数据点来训练元学习模型去完成各种相关的任务，因此新任务可以利用之前从相关任务中获得的知识，而无须从零开始训练。

(2) **少样本学习**或 **$k$ 样本学习**指的是利用较少的数据点进行学习，其中 $k$ 表示数据集各个类别中数据点的数量。

(3) 为了使模型从少量的数据点中学习，我们将用同样的方法训练它们。因此，当有一个数据集 $D$ 时，我们从出现在的数据集中的每个类别中挑选几个数据点，称之为支撑集。

(4) 我们从每个类别中挑选一些不同于支撑集的数据点，称之为查询集。

(5) 在基于度量的元学习场景中，我们将学习合适的度量空间。假设我们想找出两个图像之间的相似性。在基于度量的场景中，我们使用一个简单的神经网络从两幅图像中提取特征，并通过计算两幅图像特征之间的距离找到相似性。

(6) 我们以一种**阶段式的方式**训练模型，即，在每个阶段中，从数据集 $D$ 中抽取少量数据点，准备支撑集并在支撑集上学习。因此，在多个阶段后，模型将学会如何从较小的数据集中学习。

## 第 2 章　使用孪生网络进行人脸识别与音频识别

(1) 孪生网络是一种特殊的神经网络，也是最简单、最常用的单样本学习算法之一。孪生网络大致上由两个对称的神经网络组成，它们具有相同的权重和结构，并在最后由能量函数 $E$ 连接在一起。

(2) 对比损失函数表达式如下：

$$\text{对比损失} = Y(E)^2 + (1-Y)\max(\text{margin}-E, 0)^2$$

在上式中，$Y$ 代表真实标签（true label），当两个输入值相似时为 1，当两个输入值不相似时为 0。$E$ 是能量函数，它可以是任何距离度量。变量 margin 用于保存约束，也就是说，当两个输入值不相似时，如果它们的距离大于 margin 的值，那么就不会导致损失。

(3) 能量函数代表两个输入的相似程度，基本上可以是任何相似性度量，如欧氏距离和余弦相似度。

(4) 孪生网络的输入$(X_1, X_2)$应该成对出现，它们的二元标签 $Y \in \{0,1\}$ 代表输入对是正样本对（相同）还是负样本对（不同）。

(5) 孪生网络的应用范围相当广泛。它们可以组装在各种框架中，用于执行各种任务，比如人类动作识别、场景变化检测和机器翻译等。

## 第 3 章　原型网络及其变体

(1) 原型网络简单、高效，是目前应用最广泛使用的少样本学习算法之一。原型网络的基本思想是创建每个类的原型表示，并根据类原型与查询点之间的距离对查询点（新点）进行分类。

(2) 我们为每个数据点计算嵌入来学习特征。

(3) 一旦我们学习了每个数据点的嵌入，就可以对每个类中数据点的嵌入取平均数，并形成类原型。因此，类原型基本上是类中数据点的平均嵌入。

(4) 在高斯原型网络中，除了生成数据点的嵌入外，我们还在数据点周围添加了一个以高斯协方差矩阵为特征的置信区域。置信区域有助于描述单个数据点的质量，在数据有噪声的、同质性低的情况下十分有用。

(5) 高斯原型网络与普通原型网络的不同之处在于，在普通原型网络中，我们只学习数据点的嵌入，而在高斯原型网络中，在学习嵌入的同时，我们还为其增加了一个置信区域。

(6) 半径分量和对角分量。

## 第 4 章　使用 TensorFlow 构建关系网络与匹配网络

(1) 关系网络由两个重要的函数组成：嵌入函数 $f_\varphi$ 和关系函数 $g_\phi$。

(2) 一旦有了支撑集的特征向量 $f_\varphi(x_i)$ 和查询集的特征向量 $f_\varphi(x_j)$，我们就使用运算符 $Z$ 将它们组合起来。这里 $Z$ 可以是任意组合算子。我们使用拼接作为运算符来组合支撑集的特征向量和查询集的特征向量，即 $Z(f_\varphi(x_i), f_\varphi(x_j))$。

(3) 关系函数 $g_\phi$ 将生成0~1的关系得分，代表支撑集中样本 $x_i$ 与查询集中样本 $x_j$ 之间的相似性。

(4) 损失函数如下：

$$\varphi, \phi < -\operatorname{argmin}_{\phi,\varphi} \sum_{i=1}^{m}\sum_{j=1}^{n}(r_{i,j} - 1(y_i == y_j))^2$$

(5) 在匹配网络中，我们使用两种嵌入函数 $f$ 与 $g$，来分别学习查询集 $\hat{x}$ 和支撑集 $x_i$ 的嵌入。

(6) 查询点 $\hat{x}$ 的输出 $\hat{y}$ 的预测方法如下：

$$\hat{y} = \sum_{i=1}^{k} a(\hat{x}, x_i) y_i$$

## 第 5 章　记忆增强神经网络

(1) NTM 是一种有趣的算法，它能够在存储器中存储和检索信息，其思想是用外存储器来增强神经网络，也就是说，NTM 不是使用隐藏状态作为存储器，而是使用外存储器来存储和检索信息。

(2) 控制器基本上是一个前馈神经网络或递归神经网络。它对存储器进行读写。

(3) 读头和写头是包含存储器地址的指针，它必须从存储器中读写。

(4) 存储矩阵、存储体或简单的存储器，是我们存储信息的地方。存储矩阵基本上是由记忆细胞组成的二维矩阵，包含 $N$ 行和 $M$ 列。我们使用控制器从存储器中访问内容。因此，控制器接收来自外部环境的输入，并通过与存储矩阵交互发出响应。

(5) 基于位置的寻址和基于内容的寻址。

(6) 插值门用于决定是使用上一个时刻得到的权重 $\boldsymbol{w}_{t-1}$ ，还是使用基于内容的寻址得到的权重 $\boldsymbol{w}_t^c$ 。

(7) 从用途权向量 $\boldsymbol{w}_t^u$ 中计算最少使用权向量 $\boldsymbol{w}_t^{lu}$ 非常简单。只需将用途权向量中最小值的索引设置为 1，其他值设置为 0，因为用途权向量中最小值是最近最少使用的。

## 第 6 章　MAML 及其变种

(1) MAML 是近年来引入的、应用最广泛的元学习算法之一，在元学习研究方面取得了重大突破。MAML 的基本思想是寻找一个更好的初始参数，这样，在初始参数良好的情况下，模型可以以较少的梯度步骤快速学习新任务。

(2) MAML 与模型无关，这意味着我们可以将 MAML 应用于任何可以通过梯度下降进行训练的模型。

(3) ADML 是 MAML 的变体，它使用干净样本和对抗样本寻找更好、更稳健的初始模型参数 $\theta$。

(4) 在 FGSM 中，我们获取图像的对抗样本，计算相对于图像（拥有清晰的像素点）而不是模型参数的损失梯度。

(5) 上下文参数是在内循环中更新的特定于任务的参数 $\phi$ 。它特定于每个任务，代表单个任务的嵌入。

(6) 共享参数，记为 $\theta$，在任务间共享，并在外循环中更新，以找到最优模型参数。

## 第 7 章　Meta-SGD 和 Reptile

(1) 与 MAML 不同，Meta-SGD 不仅能找到最优的参数 $\theta$，还能找到最优的学习率 $\alpha$ 和更新方向。

(2) 学习率暗含在自适应项中。因此，在 Meta-SGD 中，不是使用小的标量值来初始化学习率 $\alpha$ ，而是使用与 $\theta$ 形状相同的随机值初始化学习率，并与 $\theta$ 一同接受学习。

(3) 学习率的更新方程如下：

$$\alpha = \alpha - \beta \nabla_\theta \sum_{T_i \sim p(T)} L_{T_i}(f_{\theta_i'})$$

(4) 抽样 $n$ 个任务，并对每个抽取的任务执行 SGD，但迭代次数更少，然后按照所有任务的共同方向更新模型参数。

(5) Reptile 更新方程：$\theta = \theta + \epsilon(\theta' - \theta)$ 。

### 第 8 章 梯度一致作为优化目标

(1) 当所有任务的梯度方向相同时，称为梯度一致。当某些任务的梯度与其他任务的梯度相差较大时，称为梯度不一致。

(2) 梯度一致更新方程：$\theta = \theta - \beta \sum_i \boldsymbol{w}_i \nabla L_{T_i}(f_{\theta_i})$

(3) 权重与任务梯度的内积以及抽样批任务中所有任务梯度的平均值成正比。

(4) 权重计算如下：

$$\boldsymbol{w}_i = \frac{\sum_{j \in T}(\boldsymbol{g}_i^T \boldsymbol{g}_j)}{\sum_{k \in T} \left| \sum_{j \in T}(\boldsymbol{g}_k^T \boldsymbol{g}_j) \right|}$$

(5) 归一化因子与 $\boldsymbol{g}_i$ 和 $\boldsymbol{g}_{\text{average}}$ 的内积成正比。

(6) 如果一个任务的梯度与抽样批任务中所有任务的平均梯度方向相同，那么可以增加它的权重，这样它对更新模型参数的贡献就会变大。类似地，如果一个任务的梯度与抽样批任务中所有任务的平均梯度有很大的不同，那么可以减少它的权重，这样它对更新模型参数的贡献就会变小。

### 第 9 章 新进展与未来方向

(1) 有很多不同类型的不平等度量：基尼系数、泰尔指数和算法的方差等。

(2) 泰尔指数是非常常用的不平等度量，以荷兰计量经济学家亨利·泰尔的名字命名。它被称为**广义熵度量**，是不平等度量家族的一个特例，并被定义为最大熵与观测熵之差。

(3) 如果让机器人通过观察我们的行为来学习，那么就可以很容易地让机器人高效地学习复杂的目标，而无须设计复杂的目标和奖惩函数。这种类型的学习，即从人类行为中学习，被称为模仿学习，即机器人试图模仿人类的行为。

(4) 概念生成器用于提取特征。可以使用由 $\theta_G$ 影响的深度神经网络来生成概念。例如，如果输入是图像，概念生成器可以是 CNN。

(5) 从任务的分布中抽取一些任务样本，通过概念生成器学习它们的概念，对这些概念进行元学习，然后计算元学习损失：

$$L_T(\theta_M, \theta_G)$$



微信连接




回复“53967”获取本书配套资源




回复“ Python ”查看相关书单



微博连接

关注@图灵教育 每日分享IT好书



QQ连接

图灵读者官方群I：218139230
图灵读者官方群II：164939616

**图灵社区**
**iTuring.cn**

在线出版，电子书，《码农》杂志，图灵访谈

“这本书对算法的解释简洁明晰、通俗易懂，任何想了解元学习的人都应该阅读。”

——**英文版读者评论**

## 什么是元学习？为什么需要了解元学习？

近年来深度学习的发展如火如荼，但深度神经网络需要大规模的训练集来训练模型，而且处理新任务时不能采用已有的模型，必须从零开始训练新模型。

元学习能够生成通用的人工智能模型来学习执行各种任务。只需少量数据点，即可训练元学习模型完成各种相关的任务。因此对于新任务，元学习模型可以利用之前从相关任务中获得的知识，无须从零开始训练。

本书介绍元学习及其原理，讲解各种单样本学习算法，并在基于Python的TensorFlow与Keras中实现它们。阅读本书，你将能够：

- 理解什么是元学习、元学习的类型及其算法
- 使用孪生网络建立人脸识别模型与音频识别模型
- 学习原型网络及其变体
- 使用TensorFlow构建关系网络与匹配网络
- 在Python中从头开始构建MAML和Reptile算法
- 掌握从头构建梯度一致算法
- 探索任务无关元学习和深度元学习


Packt>

图灵社区：iTuring.cn
热线：(010)51095183转600
分类建议：计算机 / 机器学习 / Python
人民邮电出版社网址：www.ptpress.com.cn


ISBN 978-7-115-53967-0

定价：59.00 元